MICOL MACRO

Micol

Systems

Micol Macro™ Assembler
Version 2.01

For the Apple IIGS and Apple IIe with GS upgrade by Micol Systems and Corpwell Data Systems

Micol Systems
9 Lynch Road
Toronto, Ontario
Canada M2J 2V6
(416) 495-6864

LIMIT OF LIABILITY

While every precaution has been made to the validity of the software and its accompanying manual, Micol Systems and Corpwell Data Systems cannot assume any responsibility or liability for any damage or loss caused by our software. It is the responsibility of the user to make the necessary backup for his/her data and programs.

COPYRIGHT NOTICE

DISKETTE (EXCEPT FOR ARCHIVAL PURPOSES) OR INTO MEMORY UNLESS AS PART OF THE EXECUTION OF Micol Macro™. WHEN Micol Macro™ HAS COMPLETED EXECUTION, APPLE SOFTWARE SHALL NOT BE USED IN ANY OTHER PROGRAM.

Published in Canada.

FIRST EDITION
First printing, Aug 1987.

Program and Documentation: Stephen Brunier and Allan Corupe.
Copy Editor: Ronald A. Leroux.

DISK REPLACEMENT POLICY

Our diskettes are professionally copied. However, if the original disk should prove defective within 30 days of the date of purchase, please return it with an explanation of what is wrong and a proof of purchase for prompt, free replacement or repair. If the disk has been physically damaged or if the disk fails after 90 days of the date of purchase, please include $10.00 U.S. for replacement or repair.

If failure of the product, in the judgement of Micol Systems Canada, resulted from accident, abuse, or misapplication of the product, Micol Systems Canada shall have no responsibility to replace or repair the product under the above terms.

PRODUCT REVISION

Micol Systems Canada reserves the right to make improvements to the product described in this manual at any time without notice.

The text file INFO.DOC on the master disk contains the latest information, about this product (software and reference manual), which could not be included in the manual by publication time. Load this file into the editor for up-to-date information.

UPDATE POLICY

Updated version of this software, when available, from Micol System Canada sells for $15.00 U.S. with adequate proof of purchase and a registration card on file with us. An additional $10.00 U.S. will be charged if an updated manual is to be included.

BACKUPS

The software contained on the master diskette you received with this manual is intended to be used only as a means of delivering the software, not as a work diskette on which you would do your programming. Use the copy program disk that came with your computer to make backups for working copies of the software. Keep the originals as master disks in a separate place.

We make high quality software at a reasonable price. We heartfully request that you do not abuse our policy of not copy protecting the Micol Systems' diskettes.

We do this for the convenience of the purchaser who has paid good money and should not be hindered by not being able to backup his/her software. Do make backups of this software for yourself, but do not give or lend them to others.

We would like to maintain our policy of high quality software at a reasonable price without copy protecting our diskettes. With your assistance, we will be able to do so.

SERVICE POLICY

The registration card entitles you to receive information about updates, new products, and product support. You will not be eligible for customer support or be able to have your disk updated or replaced unless you return the card.

This product has been extensively tested both by Micol Systems Canada and independent programmers. We have done everything we can to remove all the possible bugs and errors from this product.

If you should have any problems with this package, discover bugs or want to make any suggestions for improvements, then feel free to write us. Customer satisfaction is our primary goal. To improve this product, we need feedback from you. Any reasonable suggestion will be considered for future modifications and improvements.

Please send all correspondence to:

Micol Systems Canada
9 Lynch Road
Toronto, Ontario
Canada M2J 2V6
Telephone: (416) 495-6864

TABLE OF CONTENTS

3.0 THE ASSEMBLER

Appendices

PREFACE

Micol Systems is pleased to welcome you to the world of assembly language development for the Apple IIGS and the Apple IIe with GS upgrade. With this package, you have the ability to create the fastest and most versatile software your computer is able to execute.

You will be amazed at the difference in speed between programs you write with this package and BASIC programs which perform the same function. A forty time speed increase or better should be the norm. An increase of hundreds of times, although unusual, can be achieved.

Few things in life are free, and assembly language programming is no exception. While assembly language programs execute far faster than their BASIC counterparts, they are usually more difficult to write. This package will be a aid in minimizing this difficulty. The integrated Monitor/Shell, full screen text editor and automatic relocating macro assembler should be a great assistance in minimizing your programming task. You will find your investment well worth the money.

Micol Macro™ was conceived to let you make the most of the 65816 chip. Thus you will be able to write assembly language programs for the entire series of Apple II computers. We have followed, with few exceptions, the syntax rules suggested by Eyes and Lichty *(Programming the 65816 Microprocessor)* for the assembler. In addition, the *Apple IIGS Technical Reference Manual* by Michael Fischer contains invaluable information for serious software development on the Apple IIGS.

PREREQUISITES

Before you continue reading this manual, you should know

- How to set up and use your Apple II computer (see the manuals that came with your system).
- How to use ProDOS to manipulate disk files *(see the Apple II System Utilities, the ProDOS User's Disk Manual or the Apple IIGS System Disk User's Guide)*.
- The assembly language of the 65816 microprocessor.

This manual is not intended as an assembly language tutorial, but rather as a tutorial on the software contents on the diskettes you received with this package. Two books we recommend are:

- Eyes, D., and Lichty, R. *Programming the 65816: Including the 6502, 65C02 and 65802*. New York: Prentice Hill, 1986.
- Fischer, M. *Apple IIGS Technical Reference*. Berkeley, CA: Osborne McGraw Hall, 1986.

HARDWARE REQUIREMENTS

To use the Micol Macro™ Assembler, you need one of the following computer systems:

- Apple IIGS with minimum 512K RAM and one of either UniDisk (3.5 inch) micro floppy or (5.25 inch) floppy drives.
- Apple IIe with GS upgrade, minimum 512K RAM and one of either UniDisk (3.5 inch) micro floppy or (5.25 inch) drives.
- a monitor capable of displaying 80-columns.
- The Micol Macro™ Assembler operates under ProDOS 16 only, which is supplied on the Micol system disk.

SUGGESTED OPTIONS:

- a printer connected to the slot or port 1 of the IIGS or upgraded IIe.
- two or more disk drives
- a hard disk
- a RAM expansion card (a RAM disk can significantly speed up program development)

IMPORTANT NOTE

If you want to develop assembly language programs for the entire Apple II series, be forewarned of the different microprocessors used in the different Apple II computer models. Refer to Appendix C for details.

A FEW WORDS ABOUT OUR SOFTWARE

Micol Macro™ is an integrated text editor, Monitor/Shell, and self-relocating macro assembler package. The user creates his/her programs using the full-screen editor, assembles them using the macro assembler, and communicates with the system by means of the Monitor/Shell. All this instantly, without having to wait for the editor or the macro assembler to load.

The files created by the assembler are special Micol load files and can only be loaded by the loading software supplied with this product. These load files are better than the BIN files used under ProDOS 8 because they will accept multiple ORG statements and easily allow the user to create relocatable programs which will be loaded at the lowest available area of memory or to create static files which will be loaded at exactly the location the user has specified. In addition, the user can easily create SYS files which can only be loaded by ProDOS 8. SYS files must be generated if the software is to be executed on other than the Apple IIGS.

SYSTEM DISK DESCRIPTION

HOW TO RUN OUR SOFTWARE

SYSTEM DISKS SUPPLIED

Supplied with this product are two disks, a micro-floppy (3.5 inch) and a (5.25 inch) floppy disk. Because of the lack of space, the master files for Micol Macro™ are on the reverse side of the floppy disk.

PROGRAM LOADING

Place the appropriate diskette (either a 3.5 inch disk or a 5.25 inch floppy disk) with a copy of Micol Macro into the appropriate drive. Boot the computer, usually by pressing the Apple-CONTROL-Reset or turning the computer off and back on or from the ProDOS "QUIT".

If you are using a 5.25 inch floppy disk drive, when prompted, turn the disk over, re-insert it back into the drive and press the Return key. If you have a UniDisk drive, everything will load automatically. The default prefix will be set to the volume containing this software.

Micol Macro™ operates exclusively under the ProDOS 16 operating system. Those of you familiar with ProDOS on the Apple IIe and IIc (ProDOS 8) will have little difficulty, but there are some important differences you must observe.

PRODOS

Contained on the volume directory of the system disk is a file called ProDOS. Unlike the ProDOS system file under ProDOS 8 for the older Apple IIs, ProDOS is not the actual operating system, but rather an operating system loader. The actual operating system is file P16 and must reside under the subdirectory SYSTEM. If found, ProDOS loads P16 into memory. ProDOS then looks in a subdirectory of SYSTEM called SYSTEM.SETUP for the files contained in the subdirectory and

executes them. This operation initializes the system. ProDOS then goes in subdirectory TOOLS and loads these files into memory.

START

The file PRODOS then goes back to the subdirectory SYSTEM and looks for the file START. This file is the Micol preloader and simply loads and executes the file SYSTEM.LOADER under the volume directory (now the default directory). SYSTEM.LOADER is a SYS type file and resides at $2000 in bank zero.

SYSTEM.LOADER

The file SYSTEM.LOADER is necessary because ProDOS 16, the operating system cannot directly load MCL files, the load files created by the Micol Macro™ Assembler. The integrated Monitor/Editor/Assembler, the master file, is such a file. It was our feeling that the usual object module files which ProDOS 16 loader is designed to load were far too cumbersome. We therefore devised the current system.

SYSTEM.FILE

SYSTEM.FILE searches the volume directory for the file MASTER.FILE, the integrated Monitor/Editor/Assembler. MASTER.FILE is loaded and executed. You are now placed into the Monitor/Shell of Micol Macro™ ready to use the system.

MICOL SYSTEM DISK

```
PRODOS
SYSTEM.LOADER
MASTER.FILE
SYSTEM
      P16
      START
      SYSTEM.SETUP
        TOOLS.SETUP
        TOOLS
          TOOLØ21
          TOOLØ22
          TOOLxxx
```

WORK DISK

The diskette supplied with this package were not intended to be used as work disks. This is why they are write-protected (but not copy protected). It is important that you do not use the original disks for program development. Make a backup and store the originals in a separate box. Use the copy for your program development.

LOAD FILES

As long as you are with the Micol Macro™ System, by using the BRUN or BLOAD commands from the Monitor/Shell, you can easily load and execute the load files created by the macro assembler. However, once you leave this system, these files can no longer be executed directly through the operating system.

To remedy this situation, the file SYSTEM.LOADER together with the file START are designed to be used as loader files for your software independently of this system.

MASTER.FILE

Upon Execution, the file SYSTEM.LOADER runs MASTER.FILE under the default volume directory. SYSTEM.LOADER is a ProDOS 8 SYS file and always loads at $2000 in bank zero. It occupies about

3500 bytes of memory. If left undisturbed, it remains locked by the Memory Manager, and can be used as an independent loader program.

MICOL LOADER USAGE

To use this loader, once it is in memory, you simply have to input a legitimate ProDOS pathname beginning at location $2010 in bank zero. A legitimate ProDOS pathname consists of the length of the name as the first entry followed by the ASCII characters which comprise the pathname. Be certain you are writing to bank zero (use STA >$2010,X). Then simply issue a JML $2000 and the loader program will execute. Note: there are certain addresses in the SYSTEM. LOADER's direct page established at initial loading and later required by the loader. It is therefore the user's responsibility to maintain the initial value contained in the direct page register (16 bits in length). In addition, you must be certain the data bank register modes are set to your needs within the program you are loading. Do not assume any default modes.

1.0 MONITOR/SHELL

1.1 INTRODUCTION TO THE MONITOR/SHELL

The MONITOR/SHELL is the control program. Through the Monitor/Shell you can interface to the operating system, invoke the macro assembler or the text editor. Think of the Shell as being on the outside with the editor, assembler being on the inside. The MONITOR/SHELL performs the same role that the command interpreter (file BASIC.system) played under ProDOS 8. The prompt character → informs you that you are in the Monitor/Shell.

The following keys have special usage. Make use of them as they will greatly simplify the Monitor/Shell's usage. The RETURN key will not delete the characters under and to the right of it as under Applesoft.

The DELETE key

This deletes the character under the cursor, moving the characters following the cursor one place to the left.

<CTRL>R

This will repeat the previous line entered.

<CTRL>S

This will insert a space at the current cursor position, moving every character following the cursor one position to the right.

<CTRL>X

This cancels the current input.

The ↑ and ↓ arrow keys will not function and the left and right arrow keys will only function within the range of the currently entered line.

1.2 MONITOR/SHELL COMMANDS

ASSM <pathname>

This command will invoke the macro assembler and assemble the file stipulated as <pathname>. If the file mentioned in <pathname> cannot be found, you will be issued an error message and returned to the Monitor/Shell.

Normally, the assembler will append a ".B" to the pathname stipulated and use that pathname as the pathname under which the MCL file (the Micol load file) will be generated. You can override this default by specifying a comma "," followed by another pathname. The assembler will append a ".B" to the second pathname and generate the MCL file accordingly.

Example:

```
ASSM SOURCE_FILE
ASSM /RAM6/FILE,/RAM6/NEWFILE
```

BATCH <pathname>

The Batch command is used to process a batch stream through the Monitor/Shell. Pathname is the name of a text file currently on line. The file, created by the text editor, is simply a file of Monitor/Shell commands (complete with carriage returns) which you wish executed by the Monitor/Shell. Any Monitor/Shell command except another batch command, an EDIT command or an ASSM command, is a legitimate entry into this text file. Any line which begins with a semicolon(;) will be considered a comment. <CTRL>C will terminate the batch process. Batch is particularly useful to those users who are doing their development on a RAM disk and wish to set up the system as to their own requirements.

Note: There is an example batch file to create a system disk in APPENDIX A.

The commands will be displayed as they are executed.

BLOAD <pathname>

You can load an MCL file into memory by use of the BLOAD command. BLOAD will search the specified directory for <pathname>, and when found, if it is an MCL file, will load it into memory. BLOAD will inform you at which location it is loading the file. At the end of the load, you will be prompted to hit the Return key. Upon entering <RETURN>, you will be shown the machine language Monitor of the Apple IIGS with all the pertinent registers set according to the bank in which your code resides.

BRUN <pathname>

BRUN functions exactly as BLOAD except it will cause the program to execute immediately after pressing the Return key. The program will begin execution at the location generated by the last ORG statement encountered. In the case of relocatable files, this probably will not be the address as stipulated by the ORG but an address returned by the Memory Manager. In this case, the loader will inform you at which addresses it is loading the software. Hit <RETURN> when prompted to do so.

Example:

```
BRUN /RAM6/FILE.B
```

Note: The pathname stipulated under BLOAD and BRUN must be exactly as displayed in the directory, complete with .B if necessary.

CATALOG <pathname>

CATALOG and abbreviation CAT may be entered and are identical. If <pathname> is stipulated, the directory will be taken from the stipulated volume. If <pathname> does not begin with a slash "/", the default prefix will be used with the stipulated directory name. If

<pathname> is not mentioned, the directory of the default diskette will be displayed.

Example:

```
CAT /RAM6
CATALOG SUBDIR
CAT
```

CONTROL-Y

If your program BRKS to the GS monitor and you wish to return to the Micol Macro™ system, simply enter CONTROL-Y followed by a Return, and, if your program has not altered any of the system code, you will be returned to the Monitor/Shell.

COPY <pathname1> TO <pathname2>

COPY will duplicate the file of the first pathname mentioned as that of the pathname stipulated second.

Example:

```
COPY /RAM6/FILE TO /RAM7/NEWFILE
```

CREATE <pathname>

CREATE will create a new directory file under the name stipulated in the main or sub-directory as indicated by <pathname>.

Example:

```
CREATE /RAM6/DIRECT
```

DELETE <pathname>

DELETE is used to erase a file from the directory. The file must be

unlocked and the disk must not be write-protected in order to be deleted.

Example:

```
DELETE /RAM6/FILE
```

EDIT <pathname>

EDIT will invoke the CORPWELL text editor which will in turn load the file stipulated. The file must be a TXT file to be loaded. If no pathname is given and you previously had a file in memory, the last edited file will still be available for editing. (If there was no previous file, the text buffer will be cleared).

Example:

```
EDIT
EDIT /RAM6/TXT.FILE
```

FORMAT <new volume name>

If you wish to initialize a diskette or a RAM disk, then make use of FORMAT. The initialized device will have the volume name stipulated. Upon entering this command, each online volume will be displayed with its volume name, if any, prompting you to format this device or not. If either the device is accessed which contains the medium you wish initialized or the volume name appears which needs to be formatted, enter a "Y", otherwise enter a "N". Be very careful, once the "Y" is entered, any previous contents will be destroyed.

HELP

HELP will display the list of the Monitor/Shell commands available with a brief description.

Example:

```
HELP
```

HOME

HOME is used to clear the contents of the screen and place the cursor at the left corner of the screen.

Example:

```
HOME
```

LIST <pathname>

LIST displays the specified text file to the screen for the user to preview it. Only TXT type files will be displayed. Pressing <CTRL>-S will pause the listing, pressing any key thereafter will restart it. Pressing <CTRL>-C will terminate the listing.

Example:

```
LIST /RAM6/TXT.FILE
```

LOCK <pathname>

LOCK is used to protect a file from being deleted. If a file is locked, an asterisk "*" will precede the file name when taking a directory list.

Example:

```
LOCK /RAM6/FILE
```

ONLINE

ONLINE is used to determine the current online volumes.

Example:

```
ONLINE
```

PREFIX [/volume name/][{directory.name/}] [{etc.}]

PREFIX is used either to determine the default prefix the system is using or to set a different default prefix.

If <pathname> is preceded by a slash "/", it is assumed the pathname is

fully qualified. If <pathname> is not preceded by a slash "/", the default prefix will be used in front of the filename stipulated. In both cases, the system will verify that the stipulated prefix is currently an online volume. If not, the previous default prefix will remain in effect. If no pathname is stipulated, the current default prefix will be displayed.

Example:

```
PREFIX
PREFIX DIRECT/
PREFIX /RAM6/
```

QUIT

QUIT is used to terminate Micol Macro™ and 'warm boot' the system. You will first be prompted to make certain this was your intention. If "N" is entered, this command will be ignored. If "Y" is entered, control will be turned over to the operating system which will prompt you to boot the system, execute the START program or enter a new pathname. Once you have entered "Y", without rebooting the Micol master disk, you cannot return to this system.

Example:

```
User: QUIT
Computer: ARE YOU CERTAIN YOU WISH TO QUIT (Y/N) ?
User: Y
```

RENAME <pathname1> TO <pathname2>

RENAME is used to rename a file, directory file or volume name. That means, by use of this command, you can even change the directory under which the specified file resides. <pathname1> must be unlocked and <pathname2> must not already exist.

Example:

```
RENAME /RAM6/FILE TO /RAM6/NEWFILE
```

UNLOCK <pathname>

UNLOCK is the opposite of LOCK. It will unlock a file so that it may

be deleted or renamed. A space will precede the file name when the appropriate directory is displayed.

Example:

```
UNLOCK /RAM6/FILE
```

2.0 THE CORPWELL EDITOR

2.1 INVOKING/QUITTING THE EDITOR

To invoke the text editor enter "EDIT" or "EDIT" <pathname> at the Monitor/Shell level. The Corpwell Editor is always available for use since it resides in memory with the assembler/monitor. The editor has various commands that facilitate the entry and revision of assembly language source code. The commands make this editor easy to use.

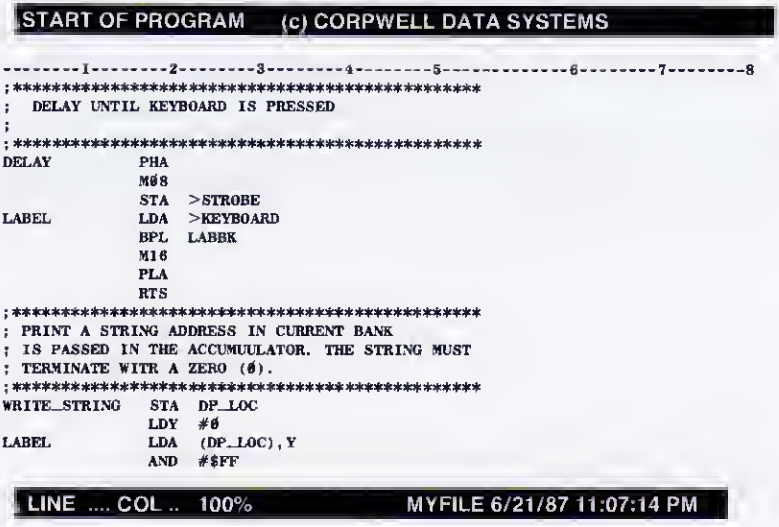
QUIT TO THE SHELL (OPTION - Q)

To quit the editor and return to the Shell, key OPTION-Q (or CLOSED-APPLE-Q *see page 12 if you have a APPLE IIe/GS retrofit).* If you made any change to your text file in memory, you will be prompted as to whether or not you wish to save the contents of the text buffer. If you respond (Y), you will be prompted for the filename. The file will be saved and the Monitor/Shell will prompt on the screen.

2.2 SCREEN INDICATORS

When the Editor is in place, you will see an inverse bar on the top of the screen. (See Figure 1). Among other functions, this line displays the Corpwell Data Systems copyright notice. The second line displays a ruler-like scale. Following the scale is the text display area. The screen displays twenty-one 80-character text lines. The bottom inverse line gives information about the position of the cursor, the amount of memory space left, the name of the file presently loaded in the text buffer and real time clock.

Figure 1.



TOP LINE

The right of the top screen line shows what command is in effect. This line also serves for input prompts from the editor.

SCALE (Ruler)

The second screen line displays a ruler-like scale. This can be used for alignment sensitive input or to speed position counting.

BOTTOM LINE

The bottom line contains the indicators about the text file you are working on.

LINE COUNTER

This count represents the cursor's current line position in the text buffer. It is affected by the ↑ and ↓ cursor movements as well as scrolling and goto line functions.

COLUMN COUNTER

Moving the cursor left or right causes the column counter to increase or decrease between 1 and 80.

MEMORY AVAILABLE

This integer percentage value indicates how much space remains in the text buffer.

FILENAME INDICATOR

This area contains a file name only after you load a file into the text buffer. This filename display remains until you save the file or clear the text buffer.

REAL TIME CLOCK

The real time clock shows the GS clock's date and time on the screen. The format is determined by the GS CONTROL Panel. When a file is saved, the date and time are automatically stamped on the file's directory information by ProDOS.

2.3 EDITOR BASICS

GETTING HELP (OPTION - ?)

The editor help screen displays which keys are used for commands. To see the Help display, press (APPLE)-? or OPTION-?. They both function the same way.

APPLE AND OPTION KEYS

On a standard Apple IIGS, the CLOSED-APPLE is replaced with the OPTION key.

Note: If your Apple IIGS is installed in a beige-colored Apple IIe retrofit, you will have to use the CLOSED-APPLE key for the OPTION key. Newer, platinum Apple IIe computers have a keyboard identical to the Apple IIGS.

ESCAPE KEY

To cancel any command in effect press escape. The escape key will not terminate a disk operation once it is in progress.

RETURN KEY

The CORPWELL EDITOR'S Return key functions much like the one of a typewriter. When the key is pressed, it places a Return symbol on the screen wherever the cursor was. The cursor then moves down to the left end of the next line of the screen. The Return key, like most of the function keys, also repeats if held down.

Note: **An assembly line must be terminated by the Return key or the assembler will think the line continues after the screen line.**

INSERT/OVERSTRIKE MODE (OPTION - E)

To set the edit mode, press OPTION-E. This toggles the insert/overstrike mode (overstrike is typing over existing characters without

inserting) from what it was previously. The default setting is Insert. Insert mode is indicated by a solid flashing cursor. Overstrike mode is shown by a flashing underscore.

DELETE CHARACTER

To delete single characters, press the DELETE key. One character will be erased and the line will adjust itself.

2.4 MOVING IN THE FILE

CURSOR CONTROLS (↑ ↓→←)

All cursor keys are functional. Pressing one of these keys once moves the cursor in the indicated direction. If one of the keys is held down, the cursor will quickly move in the direction indicated on the key. The speed and delay at which the cursor keys repeat can be regulated by the GS CONTROL Panel.

SCROLLING (↑) or (↓)

When the cursor is moved ↑ or ↓ , eventually you will reach the top or bottom of the display. When the cursor reaches the bottom, the file scrolls up. When the cursor reaches the top, the file scrolls down. The limits are TOF (top of file) and BOF (bottom of file).

PAGE SCROLLING (OPTION - ↑) or (OPTION - ↓)

Hold the OPTION and Down-Arrow key to scroll the display up one page. Alternatively, OPTION and the Up-Arrow key will scroll the text file down one screen page.

Note: you may move quickly backward or forward through a file with these commands.

GOTO LINE (OPTION - G)

To move to a specific line within your file, use the GOTO LINE command. Press OPTION-G. The top line will then prompt you for input. Give a screen line number and press Return. The line you select will be displayed on the top display line. This is a good command to use to correct errors flagged by the assembler.

Note: The Editor will GOTO a line even if the line called for is past the EOF (end of file).

START OF FILE (OPTION - 1)

To move to the start of the file, type OPTION-1.

END OF FILE (OPTION - 9)

To move to the END of the file, type OPTION-9.

TABBING

To tab, simply press the Tab key when you wish the cursor to move out to the next tab position. When you tab at the largest tab position, the cursor is moved to the first tab position on the new line. This feature is ideal for aligning the different assembly language fields.

SETTING TABS (OPTION - TAB)

You may set a maximum of 6 tabulations. To set tabs, simply press OPTION-TAB. This will display the present tab positions. They are indicated by down arrows. To set or delete tabs, move to the tab position required using the left or right cursor key, and, noting the screen rule for column position, press the space bar. A previously defined tab will be turned off. Otherwise, a tab will be set. If more than 6 tab definitions are set they will be ignored. Press Return to save your new tab selections.

DEFAULT TABS

Default tab settings are at 1, 15, 20 and 35.

2.5 BLOCK COMMANDS

COPY BLOCK (OPTION - C)

To copy a block of text, press OPTION-C. Then continue to press the ↑ or ↓ arrow key to "mark" the lines you wish to copy. The lines to be copied will be highlighted. Press the Return key once to complete the block mark. Move the cursor to where you want the marked text to be copied using the ↑ or ↓ arrow keys or page scrolling. Press Return a second time. The marked text is now **copied after** the line the cursor is on.

DELETE BLOCK (OPTION - D)

To delete a block of text, press OPTION-D. Then press the ↑ or ↓ arrow key to "mark" the lines you wish to delete. Press Return. This will delete all the lines highlighted. The text file will be refitted.

Ensure the validity of the Delete request before executing it (by pressing the Return key). There is no facility to recover a delete once completed. You can nullify the delete before execution by pressing Escape.

MOVE BLOCK (OPTION - M)

To move a block of text, press OPTION-M. Then press the ↑ or ↓ arrow key to "mark" the lines you wish to move. Press the Return key once to complete the block mark. Move the cursor to the position you want the highlighted text to be moved to using the ↑ or ↓ arrow keys or page scrolling. Press Return again and the marked text will be **moved after the line** the cursor is on.

2.6 REPLACING & FINDING

FIND STRING (OPTION - F)

To find a string in the text buffer, press OPTION-F. The top inverse editor line will prompt you for a search string. You may enter up to 64 characters for a search argument. Press Return to execute.

If the search argument is found, it is shown in inverse video in the centre of the screen. Press the addition symbol (+) to find the next occurrence or the subtract symbol (-) to locate the previous one. If you press any key other than (+) or (-), the find sequence will end.

If the argument you are searching for is not found, a NOT FOUND message will be displayed.

SEARCH & REPLACE (OPTION - R)

To search and replace a string, press OPTION-R. The top line prompt will request whether your search is to be (M)anual or (A)utomatic. Only A, M, or Escape are legal.

You are then prompted for the search string. Enter up to 64 characters followed by a Return.

A third prompt will request a replacement string. Enter up to 64 characters followed by a Return.

(M)anual S & R - If you select manual the found string will be shown in the center of the screen in inverse video. If you wish to replace it, press (Y), if not respond (N). You may press the ESCAPE key at any point to cancel a manual search and replace.

(A)utomatic S & R - If you select Automatic the editor will search and replace the string quickly without operator intervention.

Not Found - If the string you are searching for is not located in the file a Not Found message is displayed. Press any key to return to command mode.

2.7 FILING COMMANDS

CLEARING THE TEXT BUFFER (OPTION - N)

To clear the text buffer, press OPTION-N. You are prompted for confirmation. If you respond (Y)es the text buffer will be cleared.

LOAD FILE (OPTION - L)

To load a ProDOS text file into the editor, press OPTION-L. This will bring up the top prompt line allowing a 64 character pathname. Press Return to load the file. Loading a file into memory clears any previous data in the text buffer.

After the file is loaded, the editor will display 21 lines starting from line 1. The line and column counters will display 1. The percentage available value will show how much **buffer space is free** after the text file is loaded. The filename is shown on the bottom line to the left of the clock display.

If you attempt to load a new file after you have made changes to a file in memory, the editor will prompt as to whether or not you wish to save the file in memory before loading a new file.

Note: If you are attempting to load a file larger than the text buffer can hold, the file will be truncated.

SAVE FILE (OPTION - S)

To save the contents of the text buffer as a ProDOS text file, type OPTION-S. The prompt line will appear displaying the current file name (if any) Modify the filename, if necessary, and press Return. Your file is saved to the pathname specified. If you save to an existing pathname, that pathname will be deleted first, and then the new file will be created.

Note: **The assembler assembles the file from disk irrespective of the file in the editor text buffer, so be certain to make use of this command before you invoke the assembler.**

INSERT FILE (OPTION - I)

To insert/merge another ProDOS file into an existing text file already loaded, follow this procedure. Move the Cursor to the line before the position you wish the other file to be inserted. Press OPTION - I and you will be prompted for a file name.

Specify the pathname and press Return. The text starting on the line after the current cursor position will be moved to the end of the text buffer. The new text from the insert/merge file will fill the buffer until it reaches the text that is now at the end of the text buffer. If the insert/merge file is larger than the space available the overflow text will be truncated.

DISPLAY EOF MARKER (OPTION - Z)

The last line of text may not be the actual end of file position. To display the EOF mark, press OPTION-Z.

2.8 PRINT COMMANDS

Note: **Make sure your printer is turned on, is on line and has paper loaded.**

PRINT LINE RANGE (OPTION - P)

To output a range of lines to your printer, press OPTION-P. The prompt line will ask for a print range. Enter the print range in the format (startline number — endline number), separating the first and second line numbers with a hyphen. Press the Return key to initiate the printout. e.g. 100-1201.

PRINT WINDOW (OPTION - W)

To print the current text window, type OPTION-W. This command can be useful when you want to have a quick printout of the present screen display.

PAUSE PRINT (OPTION - S) or <CTRL>S

To pause output to the printer, press OPTION-S. To resume output to the printer press any key.

CANCEL PRINTOUT (ESCAPE) or <CTRL>C

To cancel the printout in effect press ESCAPE. After a cancel you will be returned to the EDITOR command level.

2.9 EDITOR/PRINTER CONFIGURATIONS

CONFIGURING TO YOUR PRINTER & INTERFACE

The APPLE IIGS has a user control panel that allows the user to customize hardware and software options, because there are so many different printers, printer interfaces and print firmware in the APPLE market place. How many characters should we allow on a line before we force a return? Should we even force a return? Likewise, should the software generate a LINE FEED after a return is sent to the printer?

To solve these problems we decided to utilize the GS control panel settings for both the number of characters allowed on a print line before a return line feed is forced, and whether or not a line feed is generated after the return. Consult your GS manual for detailed control panel usage. It should be noted here that you may enter the control panel from the SHELL or the EDITOR, make changes to the control panel and then return to the SHELL/EDITOR intact without rebooting. These control panel parameters will work with a printer I/O card or with the internal GS printer port in the same way. GS control panel Printer options other than LINE LENGTH and ADD LINE FEEDS will not effect the printout.

PRINTER LINE LENGTH

This is used to control the number of characters allowed on a printed line. The possible options are 40, 72, 80, 132 and unlimited. To set this option enter the GS control panel by entering APPLE-CNTL-ESCAPE (see your GS user manual for detailed instructions on control panel usage). In the control panel select printer options. Modify the line length indicator to one of the above selections mentioned. Exit the control panel to return back to the SHELL/EDITOR uninterrupted.

ADD LINE FEED

Most printers and interface cards allow the user the ability to add one or no line feeds to the return sequence. We felt it wise to give the option of adding a line feed to the returns. To set this option, enter the

GS control panel by keying APPLE-CNTL-ESCAPE (see your GS user manual for detailed instructions on control panel usage). Select the printer options and change the add line feed to "YES" or "NO" depending on your specific printout needs.

Note: These printer options will also effect the printer output of the assembler.

2.10 HEX & DECIMAL CONVERSION

CONVERT DECIMAL => HEX (OPTION - H)

To convert a decimal number to hex, press OPTION-H. The command line will prompt you for input. Enter the decimal number that is to be converted to hexadecimal and press the Return key. Only valid numeric characters will be allowed. Do not exceed the number 4,294, 967,296. Press any key to to continue editing.

CONVERT HEX => DECIMAL (OPTION - H)

To convert a hexadecimal number to decimal, press OPTION-H. The command line will prompt you for input. Precede the hex number with a $ (dollar sign) to indicate that the input is in hexadecimal format, then press Return. Only valid alphanumeric hexadecimal characters will be allowed. Do not exceed the value $FFFFFFFF. Press any key to restore the display.

Note: The $ (dollar sign) is what differentiates between either HEX or DECIMAL input to the number converter.

EDITOR COMMAND SUMMARY

Basic Editing

Hold down the OPTION key and press the desired key.

? - Editor Help
E - Insert/Overstrike (toggles on/off)

Moving in the File

G - Goto Line
1 - Start of file
9 - End of File
↑ - Page Scroll Up (1 page forward)
↓ - Page Scroll Down (1 page backward)
TAB - Set Tabs

Block Commands

C - Copy Block
D - Delete Block
M - Move Block

Replacing & Finding

F - Find String
R - Search & Replace

Filing Commands

L - Load File
S - Save File
I - Insert/Merge
Q - Quit to the Shell
Z - Display End of File Marker

N - Clear Buffer

Printing Commands

P - Print Line Range
W - Print Window

Miscellaneous Commands

H - Convert Decimal to Hex (toggle)

3.0 THE ASSEMBLER

3.1 INTRODUCTION TO THE ASSEMBLER

Full featured automatic relocating macro assembler

How to execute: from the Monitor/Shell, enter ASSM <pathname> or ASSM <source pathname>,<destination pathname> and the assembler will be invoked and assemble source file <pathname> contained on line.

The Micol Macro™ assembler is a full assembler, which reads as input a text file created under the text file editor, and writes as output a static, relocatable load file (type $F1 or MCL) or ProDOS SYS files (type $FF) which can later be loaded and executed by the computer.

The assembler can send listings to the screen or printer, chain and insert files, get a symbol table to dump to the screen or printer and much more. Please refer to the section on pseudo operation codes for details.

The Micol Macro™ assembler is a two pass assembler. During pass 1, the source code is read from disk and the symbol table is built. All labels are given an address and stored in the computer's memory. You can easily gauge the progress of the process because 1's are sent to the screen every 20 lines of code. During pass 2, the actual code is generated. If the LST or PRI pseudo operation is in effect, the line numbers, hexadecimal code generated by the assembler and text line will be displayed, and after the assemble is finished, the symbol table will be displayed.

During either pass, the assemble may be stopped by pressing the letter "C". During pass 2, you may pause the assembly to see the listing by pressing the letter "S". Pressing any key except "C" will start it again.

Relocatable Load Files

By proper use of the ORG pseudo operation code, as well as other pseudo operation codes and operand syntax, it is quite easy to create load files which

are relocatable; that means programs that can reside anywhere within a bank The master file containing the text editor, macro assembler and Monitor/ Shell is such a file.

In order to allow a program to be relocatable within a memory bank, it is necessary to modify the absolute addresses within the program. For relocation purposes, an absolute address is defined as any address contained within the program itself. If an address falls between the beginning of the program (the ORG statement) and the last address in the program, the address is considered absolute and must be relocated. This can be overridden by placing an underscore (_) as the first character of the label which is described later in this manual.

When the assembler encounters an address in the operand field, it first determines whether this address is absolute or not. If it is absolute, the assembler determines whether the user wishes not to relocate this address. The user can control this by careful selection of the pseudo operation codes or by use of certain syntax within the operand field. This topic will be discussed within the appropriate sections which follow.

When passing an immediate value (for example LDA #$12FF), it is assumed the value should not be altered. However, there are times when you are passing an address within your program which may need to be altered. If the instruction program contains either a ">" (for least significant byte(s)) or a "<" (for most significant byte) following the "#" for immediate addressing, and the value falls within the absolute range, then it will have its value altered accordingly.

Before the system loader loads a relocatable file, it requests from the Memory Manager the first available memory space large enough to hold this code. If you have generated a static load file, and the memory manager determines that all or part of this space is reserved, a MEMORY FULL error will be issued. Primarily for this reason, the use of relocatable load files is the method recommended both by Apple Computer, Inc. and Micol Systems. The only disadvantage to relocatable load files is that they are somewhat larger than their static counterparts and therefore require more disk space and more time to load.

However, during development, the user may wish to have fixed addresses

within his/her code, to easily determine where BRKs should be set or for other reasons. It is therefore possible to easily create load files which will load at a fixed address; no alteration will be made in the absolute addresses. The system loader also recognizes such files, but will always request the required memory from the Memory Manager before loading this file. But remember, if this memory is already occupied, the user will receive a memory full error and he/she will have to set a new address for his/her program and reassemble the source file.

Using the Memory Manager

Within the user's program itself, there are two possible ways to allocate memory for data storage, etc.

The first, and easiest method, is by the use of the RES pseudo operation code. RES will reserve the number of bytes requested within the operand field within the program space. This method is quite satisfactory for relatively small amounts of memory. However, as a load file cannot be greater than one bank of memory (64 kilobytes), this method has serious limitations (when writing a text editor for example).

For large amounts of memory or memory which must reside in an area of memory other than the one in which the code resides, the user must make use of the Memory Manager. The Memory Manager is an Apple IIGS ROM tool which can be used to allocate, deallocate, and protect memory as the user deems necessary. It's use if fully explained in the *Apple IIGS Technical Reference Manual* by Michael Fischer as well as other references.

Appendix A contains an example program which makes use of the memory manager as well as other features which the user will probably need to know. It is advised you study this program in detail.

3.2 SYNTAX OF ASSEMBLY LANGUAGE FIELDS

Comment line

Any line which begins with a semi-colon (;) in column one will be assumed to be a comment. No code will be generated from such a line, but it is recommended you make use of comments, as it will greatly aid in later maintenance of your code.

Example:

```
;This is a comment line
```

Assembly code line - 65816, 65C02/6502

This line has of a maximum of four fields: the label, mnemonic/op code or pseudo op code, address, and comment.

Example:

```
[label] mnemonic/operation [operand] [COMMENT]
[label] pseudo operation code [operand] [COMMENT]
```

These fields are described below.

1. Label field.

The label field must begin in column one and should start with a letter of the alphabet. It may contain alphanumeric characters including the underscore and be of any length. The label field is optional. A space must appear between a label and the mnemonic/operation or pseudo op code field.

If you have specified a label within an absolute address and you do not wish this address to be relocated within the load file (an address in bank zero such as $C000, for example), simply have the label begin with an underscore (_). The address will remain constant throughout the assemble.

An op code or pseudo op code must follow the label otherwise an error arises.

If you need a dummy label, use ADDRESS EQU *. Labels should not be an A, X, Y op code, a pseudo op code, or the reserved words LABEL, LABFW or LABBK.

Automatic Label Generation

The Micol Macro™ assembler is also capable of generating and accepting automatic labels. In order to use this feature, simply use the reserved word LABEL as a label. The assembler will generate successive invisible labels each time LABEL is referenced. To reference this automatically generated label, the user must make use of the reserved words LABFW and LABBK within the operand field. These label references must not contain any other characters within their operand field. To reference a previous LABEL, use LABBK (for label backward). To reference a successive label, use LABFW (for label forward). Never try to reference a point either before a previous LABEL or after a subsequent LABEL. You may have as many LABEL statements and references within your program as you require.

Be careful when using automatic labels. This feature should only be used for branches which are only a few lines away, as errors can otherwise arise.

Example:

```
        STZ NUMBER
LABEL   INC NUMBER
        LDA NUMBER
        CMP #$ØØFF
        BEQ LABFW
        BRA LABBK
LABEL   BRK $AØ
```

The BEQ LABFW will branch to the last line when executed. The BRA will branch to increment NUMBER when taken.

Needless to say, you should never use LABEL, LABBK, or LABFW as normal labels.

Local and Global Labels

The Micol Macro™ assembler lets you declare local and global labels in your source code. By default, all labels within your program are global. That is, all labels can be referenced by the entire program. If you wish, you can declare an area of your program to be local. That means that all declarations of labels in that area of code will be exclusive to that same area of code. Two labels may look the same if one is global and the other local, but they will probably have different addresses. If a label is used locally, it will be identified in the symbol table with a "#" before the label name.

A portion of your code is declared to be local if it is enclosed in "<<<" and ">>>" in the op code fields. The assembler first assumes a label is used locally and searches the symbol table for a local label. If this search fails, it searches a second time for a global label. A problem can arise if the local label was entered incorrectly. Be careful. If neither search is successful, the assembler generates an error.

Each set of "<<<" and ">>>" constitutes its own separate local area, having no relation to the previous local area. You can have a maximum of 127 local areas. It is not a good idea to use automatic labels within a local label area as potential errors may be difficult to determine.

Example:

```
COUT    EQU $FDED
        BRA MAIN      Will branch to last MAIN
STRING  STR 'This is a string'
        BYT Ø
        <<<
;Now in the local label area
COUT    EQU $FDFØ
        LDY #Ø
MAIN    LDA STRING,Y Global label used in this CASE
        BEQ *+8
        JSR COUT      Will use $FDFØ as the address
        INY
        BNE MAIN
        >>>
;Now in the global area
MAIN    LDY #Ø      Not the same main as before
```

Note: This previous example program is not intended as a guide on good pro-

gramming techniques, but merely as an example on local and global label usage.

2. Mnemonic/Op Code and Pseudo Op Field.

The op code field must follow the label with a space between these fields. If no label is used, the op code must not be in column one. All op codes and pseudo op codes are three characters long.

With the exception of DEC A and INC A, which are DEA and INA respectively, the macro assembler accepts all 65816 op codes as detailed in the Eyes/Lichty *Programming The 65816 Microprocessor* manual. (Please note that the JMP and JSR op codes cannot be used with long addresses. If you need to jump to subroutine or jump to an address greater than $FFFF, you must use JSL and JML respectively. JMP and JSR, if used in this instance, will return an error).

The pseudo operation codes are described in Section 3.3.

3. Address field

At least one space must appear between the op code and the address field. The 6502 uses 14 addressing modes, the 65C02 uses 16 addressing modes, and the 65816 uses 25 addressing modes. The Micol Macro assembler accepts all these addressing modes.

Values within the address field may consist of:

Direct page label

Absolute label

Long address label

Direct page address

Absolute address

Long address

Special characters consisting of:

*	program counter at beginning of instruction
>	least significant byte(s) or forced long addressing designator
<	most significant byte(s) or forced direct page designator
$	denotes the following value in hexadecimal
%	denotes the following value in binary
@	denotes the following value in octal
	denotes the following value in decimal (default)
+	add the following number to the previous result
-	subtract the following number from the previous result
.	(period) multiply the following number by the previous result
/	divide the previous result by the following number
#	specifies immediate addressing
'	instructs the assembler to interpret the string as APPLE modified ASCII.
!	forced absolute addressing designator

The > and < characters require special mention when used with immediate addressing. If in 8 bit mode, the > will take the least significant byte of the address passed and the < will take the most significant byte of the address. If in 16 bit mode, the > will return byte one and byte two of the address and the < will return byte two and byte three of the address.

If neither a < or a > follow the # specifying an immediate address, the assembler assumes the immediate value is a constant. If either a < or a > follow the # specifying immediate addressing, the assembler will assume the following value is an address and will alter it according to the relocation rules specified earlier.

If you are using an absolute address with PEA instruction and are generating a relocatable load file, be certain to use > and < if you wish the address to be relocated. Using a # will stop any possible relocation.

Note: The direction of the < and > in immediate addressing mode is reversed from the one stipulated by the Eyes/Lichty *Programming the 65816 microprocessor*.

4. Comment field

Following any completed line of code, you may place an optional comment. It will have no influence on the code which is generated. There must be at least one space between the previous field and the comment.

Example:

```
ADDR DEY         This is a comment
     LDA ($FF),Y This is another comment
     PHP
```

3.3 PSEUDO-OPERATION CODES

Pseudo op codes are used as instructions to the assembler. Some pseudo ops generate code, others are simply instructions to the assembler, others do both. Efficient use of these codes can make the coder's job much easier.

The pseudo operation codes fall into separate categories according to their function and usage. For this reason, we will describe them within separate categories. Be certain you read this entire section before you begin your programming.

The pseudo op codes will be described below in this order: name of the pseudo op, the description, and if necessary, an example. Each description will be followed by an error condition, if appropriate.

Pseudo Operation Codes That Effect The Symbol Table

While all operation codes can be used to effect entries into the symbol table by placing a label starting at the first column, the following pseudo operation codes are primarily intended for this purpose.

EQU

Assign the label the same value as the operand. It equates them. The operand may be a binary, octal, decimal, hexadecimal number or a label. Simple mathematics may also be used. Values may range from $0 to $FFFFFF. If a label is used, it must have been previously declared. Although it is not an error, an EQU statement without a label on the left side is worthless.

Example:

```
ADDRESS EQU $1234
```

ERROR CONDITION: It is recommended you place the EQU statements at the beginning of your program. They may be placed anywhere in the code; however, if an equate with a value of less than $0100 or a value greater than $FFFF is declared after it is referenced, subsequent addresses of labels will probably be wrong (during pass one, the assembler assumes a 3 byte opera-

tion for as yet unknown addresses). A direct page address in this instance will throw the program counter in the assembler off.

RES <number>

Reserves a number of bytes. For a fixed address file, the assembler generates <number> NOP's ($EA) and for a relocatable file the assembler simply generates a two byte number which will later be written as <number> NOPs. The number may be in any notation, even a label. It is very useful for defining relatively small variable locations.

Example:

```
MEMORY RES 5
```

Pseudo Operation Codes Generating Constant Values

The following pseudo op codes will place specified bytes of information into the object file at assembly time. You must be careful with some of these instructions as some will generate code which should be used with relocatable files, while others simply generate the stipulated values.

ABS

ABS generates a two byte value for each label in 65816 addressing format (i.e. LSB, MSB order). ABS is intended to be used to define tables of absolute addresses which will later be used within the user's program. Each label within the operand field of ABS should be an address declared within your program. ABS differs from WOR in that the value(s) within the operand field are assumed to be absolute addresses and will have these value(s) altered in the case of a relocatable load file being generated by the assembler.

Example:

```
TABLE ABS ADDRESS1,ADDRESS2,ADDRESS3
```

ASC 'a string'

ASC functions almost exactly as STR described later. ASC will generate the proper ASCII values for each character in the text string to be generated except for the last character. This means each character except the last one will have its high order bit turned off (Ø). The last character has its high order bit on (1). This makes it easy to determine the last character of the string by finding whether the value loaded is minus or not (the last character will be minus).

The string in the operand field must begin and end with a single quotation mark.

Example 1:

```
STRING ASC 'This is a text string'
```

Example 2:

```
TOOLS   EQU $E1ØØØØ
        ORG $1ØØØ
        NAT
        MØ8             8 Bit accumulator
        I16             16 Bit index registers
        BRA LABFW
STRING  ASC 'SEND THIS TEXT STRING OUT'
LABEL   LDY #Ø
LABEL   LDA STRING,Y
        PHY
        PHA
        LDX #$18ØC    Call char out from text tools
        JSL TOOLS
        PLY
        LDA STRING,Y
        BMI LABFW
        INY
        BNE LABBK
LABEL   JSR CROUT
```

BYT

Causes the assembler to generate byte values for each of the numbers appearing after the pseudo op code. You should probably limit the number of entries in a single BYT statement to a maximum of 20. The entries may be expressed as hexadecimal, decimal, octal, binary num-

ber, a single character encased in single quotation marks or a direct page label (i.e. has a value less than $0100).

No space should appear before or after a comma; anything after a space will be considered a comment.

Any legal arithmetic operation within the operand field will be accepted.

Example:

```
       WOR ADDR+5,'A',$FDEE
       BYT >LABEL,DUMMY+1
LINE   BYT 1,1,$D,$FF,>LAB1,'A'
```

ERROR CONDITION: If any value is greater than $00FF (one byte maximum value), an error condition will be flagged.

LWD <absolute addresse(s)>

LWD stands for Long Word and generates 4 bytes of code (only 2 may be displayed). This pseudo operation is designed to be used with ProDOS 16 commands or in other instances where a long address representation of an absolute address is necessary.

The code is generated in least significant byte, middle significant byte, memory bank where loaded and zero. The first two bytes will have their values relocated if necessary.

Example:

```
MLI16     EQU #$E100A8
          ORG $1000
          NAT
          M16              need 16 bit instr
          BRA LABFW
PARMLIST  RES 40
LABEL     STZ PARMLIST
          STZ PARMLIST+2
          JSL MLI16 do a Prodos 16 quit
          WOR $29
          LWD PARMLIST
          BCC LABFW
          JMP ERROR
LABEL     EQU *
```

STR 'any string'

Causes the assembler to generate one Apple ASCII character (most significant bit set (I)) for each character between single quotes. The string must be enclosed in single quotation marks. This command is extremely useful for sending output to the screen.

Example:

```
COUT    EQU $FDED
        ORG $2000     ProDOS 8 SYS file
        EMU           Want 6502 mode
        BRA PROG      NOTE: BEGIN PROGRAM EXECUTION
OUTPUT  STR 'THIS IS A STRING'
        BYT $8D
PROG    LDY #0
LABEL   LDA OUTPUT,Y
        JSR COUT
        INY
        CMP #$8D
        BNE LABBK
        BRA PROG
```

This program, when assembled and executed, will print THIS IS A STRING continuously until you hit CONTROL-Reset or turn the computer off.

WOR <value(s)>

Generates two bytes of code in 65816 addressing format (LSB, MSB). If the value is less than $0100 then MSB will be set to zero. Values may be in hexadecimal, binary, octal, decimal or a label. Absolute addresses will not be relocated with this command. If you are creating a table of absolute addresses, then use ABS. You should not specify more than 20 entries on a single source line.

No space must appear before or after a comma. Anything following a space will be considered a comment.

Example 1:

```
ADDR    EQU $EEFF
TABLE   WOR $FFFD,$FF,$1234,ADDR,0
```

The assembler will generate FD FF FF 00 34 12 FF FE 00 00.

Any legal arithmetical operation within the operand field will also be accepted.

Pseudo Operation Codes Affecting Code Generation

ORG <address>

Used to set the program counter of the assembler and cause the object code to be generated by the assembler to be written to disk. The next statement assembled will have the address specified by the ORG statement. Addresses may be specified in decimal, hexadecimal, or a previously declared label. You may have multiple ORG statements in the same program.

ORG is a very powerful statement, and with the correct use, you can cause several different types of files to be generated. It is possible to generate a ProDOS 8 SYS file, a static MCL file or a relocatable MCL file.

If the address following the ORG has a value of $2000, the assembler will generate a SYS file which can only be loaded under ProDOS 8 or by a file you create yourself. This can be useful if your code will be executed only on an Apple II+, E or C.

If the address specified following the ORG has an address greater than $FFFF, an MCL file with the fixed address specified will be created. If you wish to generate a program in bank 0, then specify an address in bank $FF (i.e. an address of $FF####). You must be careful with the address you specify because, if you try to load the file and the memory is already reserved, you will receive a memory full error from the Memory Manager. If you specify multiple ORG statements in one program, you must be certain the addresses do not overlap.

If you specify an address less than $10000, the assembler will generate a file which will be loaded at the first area available in memory (i.e. relocatable). The loader will alter all absolute addresses and related values. These files are larger than the fixed address files, but they are more flexible. You must also be very careful as to the syntax used

within a program. Some syntax will generate an address that will be relocated, while other syntax will not relocate this address. For example, the pseudo op WOR will not generate relocatable code while the pseudo op ABS will. If in doubt, please refer to the instruction description in the manual.

Note: Because of the way the assembler allocates direct page locations, it is perhaps safest not to set an ORG at less than $100.

PRG <address>

PRG causes a program to be written to memory instead of to disk. In order for code to be generated, an ORG or PRG statement must appear in the source code.

The object code is written to a buffer, then, when the assembly is complete, the code is moved to the address specified. That means no conflict should exist between your generated code and the system necessary locations. Simply do not specify an address such that your code will overwrite the assembler or the buffer area. Code not in bank 2 should be safe as the assembler/editor/monitor/shell module usually loads there.

If the buffer should overflow, the assembly will abort with the appropriate message.

Pseudo Operation Codes Affecting Macros

EXP <label> <parameter1>,<parameter2>, **etc.**

Used when you want a previously defined macro expanded at the current line. The parameters will be included in the order they are defined as in the later example.

The total number of characters involved in a macro expansion should not exceed the macro buffer area. The assembly will be aborted with the appropriate error message if the buffer overflows. This should

never be a problem as the macro area reserved on the IIGS is about 16 kilobytes.

MAC <label>

In order to define a macro command for later expansion, the programmer uses the MAC pseudo op code followed by a label. This label will be used when the macro is expanded later in the code with the EXP pseudo op. It is important the macro be defined before it is used, otherwise you will receive an error.

Parameters may be specified within the macro definition by placing a question mark (?) followed by a digit or letter, a "1" representing the first parameter, a "9" representing the ninth parameter, an "A" representing the tenth parameter, an "Z" representing the thirty-fifth parameter as if it were a base 35 system.

A TMC pseudo op terminates the macro definition and must not be left off, otherwise the macro storage area will certainly overflow.

Within the macro definition, the parameter may be designated as anything, even labels, then, when the macro is expanded, it will be changed to its definition. If the parameter is undefined, an error message will be issued, and the parameter will not be expanded.

Macros may be nested by using the **EXP** <label> statement within a macro definition. In theory, at least, no limit exists to the amount of nesting.

Example:

```
MAC EXAMPLE
LDA ?1
STA ?3        COMMENT WILL BE INCLUDED
JSR ?2        WHEN EXPANDED
TMC
EXP EXAMPLE FIRST,SECOND,THIRD
```

This will expand to:

```
LDA FIRST
STA THIRD     COMMENT WILL BE INCLUDED
JSR SECOND    WHEN EXPANDED
```

When the macro is defined, it is written to a buffer area of 16K. If this buffer should be exceeded, the assembly will be aborted with the appropriate message.

During a normal assembly, the assembler reads the source code into a buffer. If the buffer is too small, the assembler will simply process the source code until it is finished, then continue reading the rest of the code into the buffer. An area of about 16 kilobytes is reserved for macro expansion. If this is exceeded, the assembly will also abort as explained above.

TMC

This pseudo op is used to terminate a macro definition. This statement is mandatory. Without it the entire file will be considered a macro from the MAC statement to the end of file.

Pseudo Operation Codes Affecting Text Output

EJT

Used to cause a page eject (top of form). Useful only if your printer's top of form character is $C.

LST

Causes an assembled listing to be sent the the screen. Default is LST. Implies the use of an NPR pseudo op.

Pressing a letter "L" from the keyboard during pass two will also cause the listing to be sent to the screen if the screen output had been previously turned off.

NLT

Causes the listing not to be sent to the screen. Since nothing is displayed, use of NLT speeds up the assembly process considerably. If

the printer list option is also turned off, "2"s will be displayed on the CRT every 20 lines as the lines are assembled.

Pressing the letter "N" from the keyboard will also turn the listing off during pass 2.

NPR

Causes an assembled listing not to be sent to the printer. Useful to send only that portion of a listing or just the symbol table to the printer.

Pressing the letter "Q" from the keyboard during pass two will also turn off the listing to the printer.

PRI

Sends an assembled listing to the printer. If the last statement of the program is a PRI, the assembler will only send the symbol table and possible error messages to the printer.

Pressing the letter "P" from the keyboard during pass two will also turn off the listing to the printer.

The listing will be sent through port or slot 1. If you are not using the Apple IIGS' built-in serial printer port, be certain the printer's interface is in this slot, else your system will hang.

Pseudo Operation Codes Affecting Source Code Input

Normally, the assembler will read as input the source file stipulated when the assembler was first invoked from the Monitor/Shell. The following pseudo op codes are used to allow other files to be also read in. In particular, the CHN command should be used in the case of large programs (over a thousand lines of code).

CHN <pathname>

Causes the assembler to read <pathname> as if it were physically positioned after the file which was being assembled (i.e. chained).

This instruction is mostly transparent to the user. It is important that all the files to be chained are currently online. When a file is chained, a message is sent to the screen informing the user.

The assembler creates one object file (if the ORG pseudo op is used). That object file has the name of the source file (the name you typed in) with a .B appended to make certain you do not try to overwrite your text file (name can be overridden by typing a ','<pathname> following the source code name, but a .B will still be appended).

CHN should be the last line in the code. Any further lines will be ignored by the assembler.

INS <pathname>

Causes the assembler to use <pathname> as it were physically sitting at the position where the INS statement is sitting. As with EXP, line numbers will be the same throughout until the assembler is finished with <pathname>.

Be certain the needed file is currently online at assembly time.

When this statement is encountered during both passes, the statement INCLUDING <pathname> will be displayed on the screen.

Example:

```
INS /LIBRARY/FILE1
```

FILE1 from the volume /LIBRARY will be read until its end of file; its data processed one line at a time. Processing is returned thereafter to the next line in the inserting file.

A file which is being inserted may not contain an INS, EXP or CHN statement. If it does, you will get an ILLEGAL OP CODE message and the operation will be ignored.

Pseudo Operation Codes Effecting Conditional Assembly

If you wish to have source code which will be assembled at different times under different conditions, then you can make use of the following op codes.

ELS

This instruction requires no operand. It should be used only if the IFF pseudo op is still in effect. This statement will evaluate to the opposite of the IFF condition and either cause or hinder code generation accordingly. For example, if the IFF statement evaluates to greater than Ø, the ELS statement will turn off the code generation until the STP statement or the end of code.

An ELS statement not coupled with an IFF will simply turn the code generation off.

IFF <value>

There must be an operand field in this statement. If, during assembly time, the operand evaluates to a value of zero, the following statements to the STP statement, ELS statement or end of code (whichever comes first) will not be processed.

If the operand is not equal to zero, the code will be processed as if the IFF statement were not there. You must be certain the operand can be properly evaluated at the time the statement is encountered during pass 1 (i.e. no forward references), otherwise probable errors will result.

This statement is useful if you wish different code generated at different times. Conditional statements cannot be nested.

Example:

```
COND EQU Ø
     <code1>
     IFF COND
     <code2>
     ELS
     <code3>
     STP
```

By changing the operand of the EQU statement, you can cause code2 to be assembled instead of code3.

STP

This statement is used to terminate an IFF statement. It should be used only if the IFF statement is used. This statement will terminate all unresolved conditional statements in effect.

Pseudo Codes Affecting Assembler and CPU Mode.

The 65816 CPU can operate in several different modes of operation. It can operate in 6502 emulation mode, that means, as far as addressing and timing are concerned, it operates almost exactly as if it were a 65C02, the same microprocessor used in the Apple IIe and Apple IIc.

If the 65816 CPU operates in native mode, it can operate with:

- an 8 bit accumulator and memory operations and 8 bit index registers or
- an 8 bit accumulator and memory operations and 16 bit index registers or
- a 16 bit accumulator and memory operations and 8 bit index registers or
- a 16 bit accumulator and memory operations and 16 bit index registers.

There apparently is some confusion as to 8 bit mode of operation and 6502 emulation mode. While it is true the 65816 is always in an 8 bit mode if it is in 6502 emulation mode, it can also be completely in 8 bit mode when in native 65816 mode. They are two different things and must not be confused.

If the 65816 is in emulation mode, it cannot reference any memory beyond 65535 ($FFFF). For example, JML $2C000 will jump to location $C000, in the same bank. As far as long addressing is concerned, it is a 6502.

If the microprocessor is in native mode and still in 8 bit mode (the default condition when going from emulation mode to native mode), it can still reference all available memory.

From the standpoint of the developer of a macro assembler, modes of opera-

tion are the most difficult topic with which to deal. A development package should try as much as possible to protect the user from possible errors. However, in the cases of different modes such as we have here, it is impossible. It is quite possible for the assembler to generate code which, when executed, will be incorrect code. If the code is generated in one mode, but is called from a routine in another mode, the user will probably have errors which may be difficult to determine.

In order to minimize this problem, we have decided that these instructions will not only be instructions to the assembler, but will also generate the code which will cause the CPU to assume the same mode of operation.

On the surface, this seems to be a complete solution, and it would be if it were not for one important fact: the assembler will always generate code in a linear fashion, doing one line at a time. However, when the code is executed, it is seldom in a linear, sequential fashion. Branches, JSRs and JMPs create problems for this solution to be completely satisfactory. It therefore becomes the task of the programmer to be certain of the intended mode and to program accordingly.

Note: Except for the EMU and NAT pseudo ops, the code generated by the other instructions will not perform the desired task unless already in native mode. You simply cannot have 16 bit registers and memory operations if emulating a 6502. That means, if you are using this chip as a 65816, it is best that the NAT pseudo operation code be the first statement in your program.

You should also not assume any default conditions upon start of execution, but explicitly state the condition you wish to use.

EMU

If you wish the 65816 to act as a 6502, then make use of the EMU instruction. Although it will generate the code to cause the 65816 to become a 6502, the only effect this instruction has on the assembler is to cause it to ignore the operand field on the BRK operation code and generate 8 bit immediate instructions.

The assembler will generate the following equivalent code for this instruction:

```
SEC
XCE     set the emulation bit
```

I08

The I08 instruction will cause the assembler to generate 8 bit instructions for every instruction relating to the index registers using immediate addressing such as LDX #$12, and generate the code to cause the 65816 to be in 8 bit mode for the index registers.

Remember that a NAT instruction must have been encountered previously for the code generated by this instruction to have any effect on the chip.

The assembler will generate the following equivalent code for this instruction:

```
SEP #$10 sets bit 4 of the status register
```

I16

The I16 instruction will cause the assembler to generate 16 bit instructions for every instruction which relates to the index registers using immediate addressing such as LDX #$1234, and generate the code to cause the microprocessor to be in 16 bit mode for the index registers. The high order bytes of the X and Y registers will be set to zero no matter what the value those bytes may have had previously when in 16 bit mode. This is important, because if you go from 16 bit index registers to 8 bit and back to 16 bit, you will lose the high order byte of the index registers.

Remember that a NAT instruction must have been encountered previously for the code generated by this instruction to have any effect on the chip.

The assembler will generate the following equivalent code for this instruction:

```
REP #$1Ø clears bit 4 of the status egister
```

MØ8

The MØ8 instruction will cause the assembler to generate 8 bit instructions for every instruction relating to the accumulator using immediate addressing such as LDA #$12, and generate the code to cause the chip to be in 8 bit mode for the accumulator and memory instructions.

Remember that a NAT instruction must have been encountered previously for the code generated by this instruction to have any effect on the microprocessor.

The assembler will generate the following equivalent code for this instruction:

```
SEP #$2Ø sets bit 5 of the status register
```

M16

The M16 instruction will cause the assembler to generate 16 bit instructions for every instruction which relates to the accumulator using immediate addressing such as LDA #$1234, and generate the code to cause the chip to be in 16 bit mode for the accumulator. The high order byte of the accumulator will be what it was when last in 16 bit accumulator mode.

Remember that a NAT instruction must have been encountered previously for the code generated by this instruction to have any effect on the 65816.

The assembler will generate the following equivalent code for this instruction:

```
REP #$2Ø clears bit 5 of the status register
```

NAT

Will cause the assembler to require an operand field for the BRK instruction as well as generate code which will place the CPU into native 65816 mode. Although in native mode, both the accumulator and index registers will be in 8 bit mode.

Remember, this instruction is necessary to cause the microprocessor to be able to go between 8 bit and 16 bit modes.

The assembler will generate the following equivalent code for this instruction:

```
        CLC
        XCE clear the emulation bit
```

Example:

```
        ORG $1000
        NAT
        BRA MAIN
STRING  STR 'This is a demo string'
        BYT 0
MAIN    M08
        I08
        LDY #0
LABEL   LDA STRING,Y
        BEQ LABFW
        JSR COUT      Cannot be $FDED in Bank 0
        INY
        BNE LABBK
LABEL   EQU *
```

3.4 65816 ADDRESSING MODES

This section describes the different addressing modes for the different operating modes of the 65816 microprocessor necessary to understand the syntax required by this macro assembler.

There must be at least one space between the op code and the address field (or operand). Officially, there are 25 distinct addressing modes used by the 65816. However, this number seems to be somewhat of an exaggeration. Why are LDA $2000,Y and LDA $2000,X considered different addressing modes? Why does RTL have a different addressing mode than RTS when the only difference is the number of bytes pulled from the stack. Why is implied "addressing" even addressing? These are questions you will have to answer for yourself. Suffice to say, you will not have to learn 25 different syntax in order to take advantage of the addressing capabilities of either the 65816 or this assembler.

Accumulator addressing

These instructions will operate only on the accumulator. The 65816 has only four of these, they are: ASL A, LSR A, ROL A, ROR A.

Note: DEC A and INC A are treated as implied addressing by the assembler using the op codes DEA and INA respectively.

Immediate addressing

If in 8 bit mode, takes the byte following the op code as the specified address and performs its operation. If in 16 bit mode, uses the two bytes following the op code as the address and performs its operation.

Example:

```
LDA #$AØ
LDX #<LABEL
```

Note: The "#" symbol is used to specify immediate addressing.

Absolute Addressing

Uses the two bytes following the operation as its address (i.e. must be an address greater than $00FF and less than $10000). The address stated is appended to the data bank register or program bank register to yield the effective address.

According to 65816 addressing conventions, absolute addressing is expressed in least significant byte, most significant byte order. However, the assembler will take care of the order, so this should be transparent to the user.

Example:

```
LDA $FF00
LDY ADDR  LABEL IS NOT AT DIRECT PAGE
JMP $FFD2
```

Note: It is primarily this type of address which the system loader will relocate in the case the user is specifying a relocatable load file.

WARNING: This addressing mode does not necessarily stipulate memory locations within the bank in which the executing code is residing. It is the programmer's responsibility to make certain the data bank register is set according to the programmer's wishes.

Direct Page

The assembler assumes direct page addressing if the address specified after the operation code is from location $0000 to $00FF (i.e. direct page).

The actual direct page location is always to be found in bank zero. The direct page register is added to the value in the operand to give the effective address in bank 0. The direct page register is a 16 bit register which means the start of the direct page can be anywhere in bank 0.

Note: Do not confuse direct page addressing with either zero page on the 6502/65C02 or with addresses less than $0100 in the bank your code is in (even if in bank 0). If you wish to specify zero page addressing (an address less than $0100 in bank 0), precede the operand with a greater than sign ">" to force long addressing. If the operand is less than $0100 but you wish an

address in the current bank, precede the operand with an exclamation point "!" to force absolute addressing. This technique can only be used in cases where the operand field does not begin with a "(", "[" or "#".

Example:

```
LDA $FØ  direct page
LDA P1   where P1 has an address less than 256
LDA !$FØ absolute address
LDA >$FØ zero (not direct) page
```

Note: Notice the difference between direct page and immediate addressing. After the instruction LDA #$FØ is executed, the accumulator will contain an $FØ. After the instruction LDA $FØ is executed, the accumulator and direct address $FØ will have the same value.

Indexed Direct Page Addressing

Adds the content of the specified register and the byte following the operation code to the direct page register to get the address in bank Ø upon which to operate.

Example:

```
LDA $FØ,X
LDX $FØ,Y
```

Note: The assembler will consider such instructions as STA $FØ,Y as illegal because direct page addressing is impossible with the Y register. As with zero page addressing, however, you can force $FØ to be treated as an absolute address by preceding the operand with an exclamation point "!" The assembler will then accept this instruction and treat STA !$FØ,Y as an absolute instruction. If you wish zero page addressing, precede the $FØ with a greater than sign ">" to force long addressing. This technique does not work in those cases where the operand field begins with a "(", "[" or "#".

Indexed Absolute Addressing

Takes the two bytes following the operation code, adds the contents of the

specified register, appends that address to the data bank register to get the effective address.

Example:

```
LDA $ABCD,X
LDX $ABCD,Y
```

Implied Addressing

Always a one byte instruction which lacks an operand.

Example:

```
DEY
PHA
PLA
TCD
DEA
INA
```

Short Relative Addressing

The 65816 uses short relative addressing exclusively with branching. Uses the byte following the operation code as a branch offset. If bit 7 of the address byte is set, the branch will be before the operation, if not set, afterwards. The address byte is added to the program counter when the program counter is pointing to the next instruction.

With short relative branching, you may only branch about 125 bytes in either direction.

Example:

```
BRA LINE
BNE LABFW
BPL *+5
```

Long Relative Addressing

This addressing mode is identical to the short relative addressing mode except that the 2 bytes following the operation code are used as the branch offset. This makes it possible to branch relative anywhere within the bank in which the code resides.

One instruction uses this addressing mode, it is:

BRL ADDRESS

As far as the user is concerned, a BRL ADDRESS can be used instead of the JMP ADDRESS.

Direct Page Indirect

Takes the byte following the op code as an address in the direct page. The CPU fetches the value from that address and the subsequent address, appends the data bank register to that value, and uses that value as an address upon which to act.

Example:

```
LDA ($F0)
```

Let us assume that the direct page register is set to $0C00, the data bank register is set to 3, and at locations $0CF0 and $0CF1 in bank 0 are an $ FF and $C0 respectively. The CPU will first add $F0 to the $0C00 to obtain the actual address specified in bank 0. The CPU will then fetch the $FF and the $C0. It will then set the accumulator to the value contained in $C0FF in bank 3.

Indexed Indirect Addressing

Takes the value of the byte following the operation code, adds the direct page register, then adds the the contents of the X register to that value. Using that value as an address, it takes the value of that byte and the subsequent one,

appends that value onto the data bank register, and uses that value as the address upon which to perform the operation.

Example:

```
LDA ($F0,X)
```

Let us say X has a value of two, and at location $0CF2 and $0CF3 in bank 0 are $11 and $C0 respectively. The data bank register is set to 3 and the direct page register is set to $0C00. The CPU will add $0C00 to $F0 to obtain the address $CF2 in bank 0. The CPU will then fetch the $11 and $C0 obtaining $C011 as the address in bank 3. In this case only, the statement is equivalent to an LDA $3C011.

Indirect Indexed Addressing

Using the value of the byte following the operation code as a direct page address, fetches the value from the specified address and the following byte and adds the Y register to them. Then it appends that value to the data bank register in order to obtain the address on which to perform the operation.

Example:

```
LDA ($F0),Y
```

Note: A subtle but important difference exists between this addressing mode and the previous one. In the previous one, the contents of the register are added before the first address is obtained; in this case, the contents is added after the first address is obtained.

Example:

```
LDA ($F0,X) and
LDA ($F0),Y
```

Performs the same operation only if X and Y both contain zero.

Indirect Absolute Addressing

Pure indirect non-zero page addressing. The same as the previous two

instructions, except the registers are not used and the operand is an absolute address. One instruction takes advantage of this addressing mode, it is:

```
JMP (ADDR)
```

WARNING: ADDR is assumed to be an absolute address in bank Ø, so be careful. This appears to be a kludge by the microprocessor's designers.

Absolute Indexed Indirect Addressing

Takes the two bytes following the op code, adds the value contained in the X register to their contents and appends that value on to the data bank register. The CPU uses that computed value as an address which is to be acted upon. One instruction uses this addressing mode, it is:

```
JMP (ABS_ADDR, X)
```

Note: Unlike the JMP (ADDR) instruction mentioned previously, this instruction uses the bank pointed to by the data bank register for ABS_ADDR.

Stack Absolute Addressing

This addressing mode is essentially a 16 bit immediate instruction. It is always 16 bits, regardless of the accumulator or index mode.

This addressing mode is used with the PEA instruction. This instruction is used so often that you must understand the particulars of the syntax used by the assembler.

The PEA instruction is usually used to push an address onto the stack. Because the operand may or may not be an absolute address (which may be relocated at load time), you must be careful.

If the operand is preceded by a numeral sign "#", it is assumed to be a constant and will not be modified at load time.

If the operand is preceded by a greater than sign ">", it will take byte one

and byte two of the operand, and, if within the range of an absolute address, will cause this value to relocated at load time.

If the operand is preceded by a lesser than sign "<", byte two and byte three of the value will be taken, and if the value is within the range of an absolute address, at load time, byte 2 will be relocated accordingly, and byte 3 will be set to that of the bank in which the code is loaded.

Example:

```
      ORG $1ØØØ
      BRA MAIN
ADDR  RES 1Ø
MAIN  PEA #$CØØØ
      PEA >ADDR  2 LSB's, relocatable
      PEA <ADDR  2 MSB's, MSB will be load blank #
      PEA #>ADDR fixed address
```

Absolute Long Addressing

Absolute long addressing is the same as absolute addressing except that the program or data bank is explicitly specified in the address field. This means that the value in the address field must be greater than $FFFF. If a label is used within the operand field, it must have been declared before this statement is encountered or errors in assemble will result. Unless the operation is a JML or JSL, during pass one, the assembler assumes a three byte instruction for operand fields not yet defined. Since this addressing mode requires a 4 byte instruction, **it is mandatory that labels within this field be previously declared.**

If the value in the operand field is $FFFF or less and you wish to use absolute long addressing, then specify a > at the beginning of the field.

Example:

```
LDA $34567 $4567 in bank 3
STA >$CØØØ $CØØØ bank Ø
JML $2AØØØ note!! must use JML and not JMP
JSL $31234 note!! must use JSL and not JSR
```

Absolute Long Indexed with X

This addressing mode is identical to absolute long addressing except the value contained in the X register is added to the address in the operand field in order to obtain the effective address.

Example:

```
LDA $12345,X
LDA LONG_ADDRESS,X
STA >$2000,X  address in bank 0
```

Direct Page Indirect Long Addressing

This addressing mode is identical to direct page indirect addressing mode except the value contained in the byte following the direct page address specified is used to specify the bank instead of the data bank register.

Example:

```
LDA [$F0]
STA [DIRECT_PAGE]
```

Stack Direct Page Indirect

This addressing mode is identical to direct page indirect except for the effect it has on the contents of the stack while this operation is being executed.

This addressing mode adds the value contained in the operand field to the 16 bit direct page register and pushes that 16 bit value onto the stack.

Example:

```
PEI (DIRECT_PAGE)
```

Stack Program Counter Relative

Adds the value of the 16 bit program counter at the beginning of the next

instruction to the signed address specified in the operand field and pushes that value onto the stack.

Example:

```
PER ADDRESS
```

Stack Relative Addressing

Adds the value of the one byte operand to that of the stack pointer to get the address in bank Ø.

Example:

```
LDA 3,S fetch the third (and maybe forth) byte(s) after SP
```

Stack Relative Indirect Indexed

This addressing mode is somewhat similar to both the stack relative and direct page together. The indirect address is accessed the same way as the value is accessed in the stack relative. This 16 bit value is then used as an address where the required data is accessed.

Example:

```
LDA (1,S),Y
```

Block Move

The block move addressing mode is by far the most complicated one to understand as well as to implement as the registers must contain certain values as well as the specification of tbe operand field.

The 16 bit accumulator must contain the number of bytes minus one to be moved. The sixteen bit X register contains the starting address of the source. The sixteen bit Y register contains the starting address of the destination. The first operand field must specify the source bank and the second operand instruction must specify the destination bank.

In Micol Macro™, the assembler expects to find the complete source and destination addresses in the operand field. This is done to make the coding easier for the programmer; the assembler will ignore all but the bank number.

Example:

```
M16                 must have 16 bit registers
I16
LDA #$2001          move $2000 bytes
LDX #$8000          source is at $8000
LDY #$2000          destination is at $2000
MVN $28000,$42000 move from bank 2 to bank 4
```

Note: This instruction is probably not as fast as you might think. It requires seven clock cycles for each byte moved. Also, in tests we have run, the two instructions using this addressing mode MVN and MVP, were not able to move data across bank boundaries, significantly limiting their usefulness.

3.5 ASSEMBLER ERROR MESSAGES

When errors occur, the message(s) will be flagged by the assembler as they are found, printed in inverse. In addition, a summary of the errors will appear after the symbol table dump.

ADDRESSING ERROR AT <row number> IN FILE [/volume.name/] filename

The address field of the specified line is in error.

DOUBLE DECLARATION ERROR AT <row number> IN FILE [/volume.name/] filename

You have declared the same label a second time. Appears during pass one.

OP CODE ERROR AT <row number> IN FILE [/volume.name/] filename

The op code used was not recognized as a 65816 instruction or as a Micol Macro's™ pseudo op code.

UNKNOWN LABEL ERROR AT <row number> IN FILE [/volume.name/] filename

You have referenced a label that was never declared.

BRANCH OVERFLOW ERROR AT <row number> IN FILE [/volume.name/] filename

This error can have two causes:

a) the label to which you are branching has not been declared.

b) you have tried to branch more than the allowable bytes (about 126 bytes in either direction).

SYMBOL TABLE OVERFLOW AT <row number>
You have declared more labels than there is memory to store them.

FATAL ERROR: ABORT, HIT KEY, BUFFER OVERFLOW AT <row number>

If the source code buffer during a macro expansion, the macro buffer during

a macro definition or the memory buffer as the result of a PRG statement should overflow, the assembly process will simply be aborted with this message HIT ANY KEY TO CONTINUE. Because the Apple IIGS has a large amount of memory, this error will probably never occur.

At the conclusion of the assembly process, after the symbol table dump, the error messages will be displayed again. Pressing the letter "S" will pause the list for you to read (press any key except "C" to continue). Pressing the letter "C" will terminate the error messages.

The system stops counting or storing errors found during assembly if greater then 127, but will continue to report them. The line numbers used are the same as used by the editor.

Symbol Table Dump

At the conclusion of the assembly, all labels and their addresses will be displayed if LST or PRI is in effect. If a label has not been accessed (has no use in the code), it will be displayed in inverse on the screen and have a ">" pointing to its address if printed. A local label will be designated by a "#".

The symbol table and the Apple IIGS monitor will probably be your most powerful debugging tools.

3.6 HOW TO ASSEMBLE YOUR PROGRAMS

1. You must first have created and saved your program source file using the text editor.

2. Specify ASSM <pathname> or ASSM <source pathname>, <destination pathname>.

3. The assembler will process your file, generating 65816 code to a file if disk, or a buffer if the PRG pseudo op has been used, saving the code to the name specified above with a ".B" appended assuming the ORG pseudo op was used in the program.

4. When finished, you will get the error messages, if any, giving you information about the total number of errors, number of bytes of code generated, and number of lines processed. If you have used macros or inserted a file, the last figure will probably be greater than the actual number of lines of text you had edited.

5. If you have specified the PRG pseudo op in your code, the object code will be moved from its buffer to the area specified. You will then be asked if you wish to execute it. Press "Y" if you do and "N" if not. If "Y" is pressed, the execution will begin at the first byte of code, so be certain your code's execution begins at the first byte. If your code has not overwritten any of the assembler in memory, you may return to the command level of the assembler by pressing <CTRL> Y <CR> after a BRK has been executed in your program.

If you have specified an ORG instead of PRG in your program (the method we strongly recommend) and have received no syntax errors, you will receive the prompt B(load), R(un), M(onitor/shell). If you wish your assembled file loaded, Press B. After your file has loaded, you will be placed into the GS monitor. If you wish to execute your assembled file, press R. Your file will be loaded and execution will begin at the last ORG statement stipulated. If you press M, you will be returned to the Monitor/Shell. Only these three letters are acceptable input.

APPENDIX A

- EXAMPLE PROGRAM -

```
BUF_HANDLE EQU  $10
BUF_MEMORY EQU  BUF_HANDLE+4
ACC_REG        EQU  BUF_MEMORY+4
X_REG          EQU  ACC_REG+2
Y_REG          EQU  X_REG+2
DP_LOC         EQU  Y_REG+2
KEYBOARD       EQU  $C000
STROBE         EQU  $C010
TOOLS          EQU  $E10000          ROM TOOLS ENTRY POINT
;*******************************************************
; GET MEMORY FROM THE MEMORY MANAGER MACRO.
; FIRST DEFINE THE MACRO WHICH WE WILL EXPAND
; LATER SEVERAL TIMES.
;*******************************************************
; THE ATTRIBUTE BYTE DETERMINES
; WHERE AND HOW MEMORY IS ALLOCATED BY THE
; MEMORY MANAGER; ITS USE IS CRITICAL.
; THE BIT CONFIGURATIONS (GOING FROM MOST SIGNIFICANT
; BYTE TO LEAST SIGNIFICANT BYTE) ARE:
;
; BIT 15:      1 = MEMORY WILL BE LOCKED, 0 = UNLOCKED
; BIT 14:      1   CANNOT BE MOVED, 0 = CAN BE MOVED
; BIT 13-11:    (NOT USED)
; BITS 8-9:    3 = HIGHEST, 0 = LOWEST PURGE LEVEL
; BITS 5-7      (NOT USED)
; BIT 4:       1 = MAY, 0 = MAY NOT CROSS BANK BOUNDRIES
; BIT 3:       1 = MAY, 0 = MAY NOT USE SPECIAL MEMORY
; BIT 2:       1 = DO,  0 = DO NOT PAGE ALIGN
; BIT 1:       1 = SPECIFIED ADDRESS, 0 = RELOCATABLE ADDRESS
; BIT 0:       1 = FIXED, 0 = ANY BANK
;*************************************************

              MAC  MEMORY_MANAGER
              PEA  #0
              PEA  #0               SPACE FOR HANDLE
              PEA  #0
              PEA  #?1              SPECIFY NUMBER OF BYTES
              LDA  SYSTEM_ID        ID RETURNED FROM MMSTARTUP
              PHA
              PEA  #?2              ATTRIBUTE BYTE
              PEA  #0               LONG ADDRESS (IF APPLICABLE)
              PEA  #0
```

1 = MAY NOT

```
                LDX  #$0902
                JSL  TOOLS
                BCC  *+5
                JMP  ERROR
                PLA                     GET THE HANDLE
                STA  0
                STA  ?3                 PARM 3 MUST BE THE HANDLE VAR
                PLA
                STA  2
                STA  ?3+2
                LDA  [0]
                STA  ?4                 PARM 4 MUST BE THE MEMORY VAR
                LDY  #2
                LDA  [0],Y
                STA  ?4+2
                TMC
;***********************************************
; TO RELEASE THE MEMORY BUFFERS.
; LET'S USE THE SAME TECHNIQUE AS BEFORE,
; FIRST DEFINE A MACRO WE WILL USE SEVERAL
; TIMES, AND THEN DEFINE IT WITH THE
; PARAMETERS WE REQUIRE.
;***********************************************
                MAC  RELEASE
                LDA  ?1+2
                BEQ  LABFW              USE MSB AS RELEASE FLAG
                PHA
                LDA  ?1
                PHA
                LDX  #$1002
                JSL  TOOLS
                BCC  *+5
                JMP ERROR
                STZ  ?1+2
LABEL           RTS
                TMC
                ORG  $1000              MAKE A REL FILE
                NAT                     WANT 65816 NATIVE MODE
                M16                     WANT A 16 BIT ACCUMULATOR
                I16                     WANT 16 BIT INDEX REGISTERS
                JMP  START
;***********************************************
;       THE FOLLOWING STRINGS ARE THE COMPLETE SET
;       OF ERROR MESSAGE FOR THE MEMORY MANAGER
;***********************************************
MESSAGES        BYT  1
                STR  'Memory full error'
                BYT  0,2
```

```
                STR  'Illegal operation on a ''NIL'' handle'
                BYT  Ø,3
                STR  '''NIL'' handle expected for this operation'
                BYT  Ø,4
                STR  'Illegal operation on a locked or fixed blk'
                BYT  Ø,5
                STR  'Attempt to purge an unpurgeable block'
                BYT  Ø,6
                STR  'Invalid handle given'
                BYT  Ø,7
                STR  'Invalid owner ID given'
                BYT  Ø,8
                STR  'Illegal load operation code'
                BYT  Ø,$FF
SYSTEM_ID       RES  2              NOT IN DP BECAUSE WILL CHANGE
PRTEMP          RES  2              TEMP LOCATION FOR PRBYTE
DP_HANDLE       RES  4              CANNOT BE AT DIRECT PAGE
DP_MEMORY       RES  4              BECAUSE THE DP REG WILL CHANGE
ERR_VALUE       RES 2
HITPROMPT       STR  'HIT ANY KEY TO CONTINUE'
                BYT  $8D,$8D,Ø
DETAILS_OUT     STR  ' IS BEGINNING OF MEMORY ALLOCATION'
                BYT  $8D,Ø        Ø IS A GOOD STRING END
HIT             JSR  CROUT
                JSR  CROUT
                LDY  #Ø
LABEL           LDA  HITPROMPT,Y
                AND  #$FF
                BEQ  LABFW
                JSR  COUT
                INY
                CPY  #$FF           FAILSAFE TEST
                BNE  LABBK
LABEL           JSR DELAY
                RTS
BELL            PHA                 NEED TO KEEP C REG. SAFE
                LDA  #7
                JSR  COUT
                PLA
                RTS
ERROR           JSR  BELL
                AND  #$FF           DON'T WANT MSB OF ERROR CODE
                STA  ERR_VALUE
                JSR  HOME
                LDA  ERR_VALUE
                LDY  #Ø
LABEL           LDA  MESSAGES,Y     FIND START OF ERROR MESSAGE
                AND  #$FF           KEEP ONLY ONE CHARACTER
                CMP  #$FF           (I.E. ILLEGAL MESSAGE?)
```

```
                BEQ   ERROR100          NO MESSAGE FOUND?
                CMP   ERR_VALUE
                BEQ   ERROR200          MATCH?
                INY
                CPY   #$FF
                BNE   LABBK
ERROR100        BRK   $F0                AN ERROR IN PRINTING MESSAGE ?
ERROR200        LDA   MESSAGES+1,Y      NOW PRINT MESSAGE
                AND   #$FF
                BEQ   LABFW
                JSR   COUT
                INY
                BNE   ERROR200
LABEL           JSR   HIT
                BRK   $F1               USER CAN DO WHAT HE WANTS HERE
SAV_REG         STA   ACC_REG
                STY   Y_REG
                STX   X_REG
                RTS
RES_REG         LDA   ACC_REG
                LDY   Y_REG
                LDX   X_REG
                RTS
COUT            AND   #$FF              MUST HAVE ONLY ONE BYTE
                JSR   SAV_REG
                LDA   >KEYBOARD         HAS A KEY BEEN PRESSED?
                BPL   COUT100
                CMP   #$93              WAS THE KEY <CTRL> S
                BNE   COUT100
LABEL           STA   >STROBE           YES, SO DELAY UNTIL KEYPRESS
LABEL           LDA   >KEYBOARD         PAUSE DESPLAY
                BPL   LABBK
                STA   >STROBE
COUT100         LDA   ACC_REG
                CMP   #$8D              IF A <CR>, FORCE A <LF> ALSO
                BNE   LABFW
                LDA   #$8A              IS PASCAL OUTPUT, NEED <LF> ALSO
                PHA
                LDX   #$180C
                JSL   TOOLS
LABEL           LDA   ACC_REG           PRINT OUT THE CHARACTER
                PHA
                LDX   #$180C
                JSL   TOOLS
                JSR   RES_REG
                RTS
```

```
CROUT           PHA
                LDA   #$8D
                JSR   COUT
                PLA
                RTS
;***********************************************
;  DELAY UNTIL KEYBOARD IS PRESSED
;            ANY KEY WILL DO
;***********************************************
DELAY           PHA
                MØ8
                STA   >STROBE
LABEL           LDA   >KEYBOARD
                BPL   LABBK
                M16
                PLA
                RTS
;***********************************************
; PRINT THE STRING WHOSE ADDRESS IN CURRENT BANK
; IS PASSED IN THE ACCUMUULATOR. THE STRING MUST
; TERMINATE WITH A ZERO (Ø).
;***********************************************
WRITE_STRING    STA   DP_LOC
                LDY   #Ø
LABEL           LDA   (DP_LOC),Y
                AND   #$FF
                BEQ   LABFW
                JSR   COUT
                INY
                BNE   LABBK
LABEL           RTS
;***********************************************
; ID TAG MANAGER
; CALLED ONLY IF MMSTART GIVES AN ERROR
;***********************************************
GETNEWID        PEA   #Ø
                PEA   #$13ØØ
                LDX   #$2ØØ3
                JSL   TOOLS
                BCC   MMSTARTUP100
                JMP   ERROR
;***********************************************
; START UP THE MEMORY MANAGER
; GET THE APPLICATION ID
;***********************************************
MMSTARTUP       PEA   #Ø
                LDX   #$Ø2Ø2
                JSL   TOOLS
                BCC   MMSTARTUPlØØ
```

```
                CMP   #$207           INVALID OWNER ID ERROR
                BNE   LABFW
                PLX
                BRA   GETNEWID
LABEL           JMP   ERROR
MMSTARTUP1ØØ    PLA
                STA SYSTEM_ID
                RTS
;***********************************************
; INITIALIZE THE TEXT DISPLAY TO 80 COLUMNS PASCAL
;***********************************************
INIT_SCREEN     PEA   #1              TURN ON 80 COLUMNS
                PEA   #Ø
                PEA   #3              SLOT 3
                LDX   #$1ØØC          TEXT TOOL CODE
                JSL   TOOLS
                PEA   #Ø              INITIALIZE OUTPUT
                LDX   #$15ØC
                JSL   TOOLS
                RTS
;***********************************************
; TERMINATE THE ENTIRE APPLICATION
;***********************************************
MMAPPQUIT       LDA SYSTEM_ID
                PHA
                LDX #$3Ø2
                JSL TOOLS
                BCC LABFW
                JMP ERROR
LABEL           RTS
;***********************************************
; MAKE A SAFE DIRECT PAGE LOCATION
; THIS SHOULD BE THE SECOND ROUTINE
; CALLED, AFTER MMSTARTUP
;***********************************************
GET_DIR_PAGE    EXP   MEMORY_MANAGER $1ØØ,$$CØØ1,DP_HANDLE ,DP_MEMORY
                PEA   #Ø
                PEA   #Ø              SPACE FOR HANDLE
                PEA   #Ø
                PEA   #$1ØØ           SPECIFY NUMBER OF BYTES TO GET HERE
                LDA   SYSTEM_ID       ID RETURNED FROM MMSTARTUP
                PHA
                PEA   #$CØØ1          ATTRIBUTE BYTE
                PEA   #Ø              LONG ADDRESS(IF APPLICABLE)
                PEA   #Ø
                LDX   #$Ø9Ø2
                JSL   TOOLS
                BCC   LABEL
                JMP   ERROR
```

```
LABEL           PLA                     GET THE HANDLE
                STA  Ø
                STA  DP HANDLE          PARM 3 MUST BE THE HANDLE VAR
                PLA
                STA  2
                STA  DP HANDLE+2
                LDA  [Ø]
                STA  DP MEMORY          PARM 4 MUST BE THE MEMORY VAR
                LDY  #2
                LDA  [Ø],Y
                STA  DP MEMORY+2
                TMC
;NOW SET YOUR DIRECT PAGE
                LDA  DP_MEMORY          SET THE DIRECT PAGE MEMORY
                TCD
                RTS
;***********************************************
; GET A BLOCK OF MEMORY FROM ANY LOCATION
; IN THIS EXAMPLE WE WILL ALLOCATE $8ØØØ BYTES
;***********************************************
GET_BUFFER      EXP  MEMORY_MANAGER $8ØØØ,$C3ØØ,BUF_HANDLE,BUF_MEMORY
                PEA  #Ø          [CODE TO TMC EXPANDED BY ASSEMBLER]
                PEA  #Ø                 SPACE FOR HANDLE
                PEA  #Ø
                PEA  #$8ØØØ             SPECIFY NUMBER OF BYTES
                LDA  SYSTEM_ID          ID RETURNED FROM MMSTARTUP
                PHA
                PEA  #$C3ØØ             ATTRIBUTE BANK
                PEA  #Ø                 LONG ADDRESS(IF APPLICABLE)
                PEA  #Ø
                LDX  #$Ø9Ø2
                JSL  TOOLS
                BCC  *+5
                JMP  ERROR
                PLA                     GET THE HANDLE
                STA  Ø
                STA  BUF_HANDLE  PARM 3 MUST BE THE HANDLE VARIABLE
                PLA
                STA  2
                STA  DP HANDLE+2
                LDA  [Ø]
                STA  BUF_MEMORY  PARM 4 MUST BE THE MEMORY VARIABLE
                LDY  #2
                LDA  [Ø],Y
                STA  BUF_MEMORY+2
                TMC
                RTS
```

```
;***********************************************
; DISPLAY THE ADDRESSES OBTAINED
;***********************************************
DISPLAY_ADDR  LDA #'$'            SHOW THE VALUE IS IN HEX
              JSR  COUT
              LDA  DP_MEMORY+2
              JSR  PRBYTE
              LDA  DP_MEMORY+1
              JSR  PRBYTE
              LDA  DP_MEMORY
              JSR  PRBYTE
              LDA  #>DETAILS_OUT
              JSR  WRITE_STRING
              JSR  CROUT
              LDA  #'$'
              JSR  COUT
              LDA  BUF_MEMORY+2
              JSR  PRBYTE
              LDA  BUF_MEMORY+1
              JSR  PRBYTE
              LDA  BUF_MEMORY
              JSR  PRBYTE
              LDA  #>DETAILS_OUT
              JSR  WRITE_STRING
              JSR  HIT
              RTS
;***********************************************
; NOW CREATE CODE WHICH WILL RELEASE THE CODE
;***********************************************
RELEASE_DP    EXP  RELEASE DP_HANDLE
              LDA  DP_HANDLE+2
              BEQ  LABFW          USE MSB AS RELEASE FLAG
              PHA
              LDA  DP HANDLE
              PHA
              LDX  #$1002
              JSL  TOOLS
              BCC  *+5
              JMP ERROR
              STZ  DP_HANDLE+2
LABEL         RTS
LABEL         TMC
RELEASE_BUF   EXP  RELEASE BUF_HANDLE
              LDA  BUF_HANDLE+2
              BEQ  LABFW          USE MSB AS RELEASE FLAG
              PHA
              LDA  BUF_HANDLE
```

```
                PHA
                LDX  #$1002
                JSL  TOOLS
                BCC  *+5
                JMP ERROR
                STZ  BUF_HANDLE+2
LABEL           RTS
LABEL           TMC
;***********************************************
; SAME AS A HOME YOU ARE USED TO
;***********************************************
HOME            PHA
                LDA  #$C
                JSR  COUT
                PLA
                RTS
;***********************************************
; SEND OUT THE VALUE IN THE LEAST SIGNIFICANT
; BYTE OF THE ACCUMULATOR IN ITS HEXADECIMAL NOTATION
;***********************************************
PRBYTE          STA  PRTEMP
                AND  #$FF           GET RID OF MSB FOR NOW
                LSR  A
                LSR  A              ISOLATE MOST SIGNIFICANT NIBBLE
                LSR  A
                LSR  A
                CMP  #$0A           IS DIGIT OR LETTER ?
                BCC  LABFW
                CLC
                ADC  #$07
LABEL           ADC  #'0'           MAKE IT AN ASCII VALUE
                JSR  COUT
                LDA  PRTEMP
                AND  #$0F
                CMP  #$0A
                BCC  LABFW
                CLC
                ADC  #$07
LABEL           ADC  #'0'
                JSR  COUT
                LDA  PRTEMP         RESTORE ORIGINAL VALUE
                RTS
;PROGRAM EXECUTION STARTS HERE
START           JSR  MMSTARTUP      INIT MEMORY MANAGER, GET ID
                JSR  GET_DIR_PAGE   GET DIRECT PAGE MEMORY IN BNK 0
                JSR  GET_BUFFER     GET LARGE BUFFER IN ANY BANK
                JSR  DISPLAY_ADDR
                JSR  INIT_SCREEN
```

```
JSR  RELEASE_BUF     RELEASE MAIN BUFFER
JSR  RELEASE_DP      RELEASE DIRECT PAGE MEMORY
JSR  MMAPPQUIT       QUIT THIS APPLICATION
BRK  $1Ø
```

Example of a Batch Program

The following batch file can be used to create a Micol Macro system disk from the master disk supplied to a RAM card with sufficient space in any slot. The user should make the necessary modifications for his/her own needs.

Enter the following text using the Corpwell Editor making the necessary modifications; then save it under any suitable file name. QUIT the editor to the Monitor/shell and enter BATCH < batch filename<CR>>. The computer will take control.

```
;NOTE: FORMAT WILL REQUIRE USER INPUT
FORMAT /RAM.DISK
PREFIX /MICOL.MACRO
COPY   PRODOS TO /RAM.DISK/PRODOS
CREATE /RAM.DISK/SYSTEM
PREFIX /MICOL.MACRO/SYSTEM
COPY   P16 TO /RAM.DISK/SYSTEM/P16
COPY   START TO /RAM.DISK/SYSTEM/START
CREATE /RAM.DISK/SYSTEM/SYSTEM.SETUP
PREFIX /MICOL.MACRO/SYSTEM/SYSTEM.SETUP
COPY   TOOLS.SETUP TO /RAM.DISK/SYSTEM/SYSTEM.SETUP/TOOLS.SETUP
CREATE /RAM.DISK/SYSTEM/TOOLS
PREFIX /MICOL.MACRO/SYSTEM/TOOLS
;USER MUST DETERMINE WHICH TOOLS HE WISHES TO COPY
COPY   TOOLØ2Ø TO /RAM.DISK/SYSTEM/TOOLS/TOOLØ2 Ø
COPY   TOOLØ22 TO /RAM.DISK/SYSTEM/TOOLS/TOOLØ22
PREFIX /RAM.DISK
COPY   /MICOL.MACRO/MASTER.FILE TO MASTER.FILE
```

APPENDIX B

- SYNTAX DEFINITIONS -

< > enclose a description

[ ] enclose elements that are optional

/volume.name : = indicate any legal volume.name

sub-directory/ : = indicate any legal sub-directory name

filename : = indicate any legal filename

/prefix/ : = /volume.name/sub-directory1/sub-directory-N/

Spaces are included for clarity.

APPENDIX C

- APPLE II MICROPROCESSOR LIST -

This appendix lists the processors used in the entire Apple II series of computers with approximate beginning and ending of the manufacturing date of each model.

Model	Processor	Operating Syst.	Manuf. Dates
Apple II	6502	DOS 3.2	April 77 - May 79
Apple II **	6502	Dos 3.3	June 78 - Dec 82
Apple IIe *	6502	DOS 3.3	Jan 83 - Feb 85
		ProDOS	March 84 -
Apple IIe	65C02	ProDOS	March 85 - Feb 87
Apple IIe	65C02	ProDOS	March 87 -
Apple IIc	65C02	ProDOS	April 84 -
Apple IIGS	65816	ProDOS	Sept 86 -

* The Apple IIe Enhancement Kit will update these machines. It will then have a 65C02 processor and the same ROMs as the Apple IIe enhanced.

** Needs language card to run ProDOS.

APPENDIX D

- RESERVED WORDS -

- (PSEUDO OP'S) -

- SPECIAL FEATURES-

PSEUDO	FIELD	COMMENT
<<<	op code	Beginning of local labels
>>>	op code	End of local labels
LABEL	label	Mark an automatic label
LABBK	operand	Reference a previous LABEL
LABFW	operand	Reference a successive LABEL
ABS	op code	Used to make a table of absolute address
ASC	op code	Used to declare an ASCII string with a terminal mark
BYT	op code	Used to declare byte values within a program
CHN	op code	Chain the specified program file
EJT	op code	Sends a top of form to the printer
ELS	op code	Perform the opposite of the conditional assembly operation
EMU	op code	Specify emulation mode to the CPU and the assembler
EQU	op code	Assign the operand value to the label
EXP	op code	Expand the referenced macro
I08	op code	Set index register mode to 8 bits
I16	op code	Set index register mode to 16 bits
IFF	op code	Conditional assembly operative
INS	op code	Include the specified file
LST	op code	Send assembled listing to screen
LWD	op code	Generate a 4 byte value for an absolute address
M08	op code	Set 8 bit accumulator mode

M16	op code	Set 16 bit accumulator mode
MAC	op code	Start a macro declaration
NAT	op code	Set 65816 native mode
NLT	op code	Stop sending the assembled listing to the screen
NPR	op code	Stop sending the assembled listing to the printer
ORG	op code	Set PC of assembler, write code to disk
PRG	op code	Set PC of assembler, write code to a buffer
PRI	op code	Send assembled listing to the printer
RES	op code	Reserve specified number of bytes
STP	op code	Terminate an IFF or ELS pseodo op
STR	op code	Used to declare an Apple ASCII string
TMC	op code	Used to terminate a macro definition
WOR	op code	Used to declare a 16 bit value in LSB, MSB order

APPENDIX E

- EDITOR COMMAND SUMMARY (Alphabetical) -

Hold the OPTION key and press the desired key.

C	- Copy Block
D	- Delete Block
E	- Insert/Overstrike (toggle)
F	- Find String
G	- Goto Line
H	- Convert Hex/Decimal Numbers (toggle)
I	- Insert/Merge File
L	- Load File
M	- Move Block
N	- Clear Buffer
P	- Print Range
Q	- Quit to Shell
R	- Search & Replace
S	- Save File
V	- Version Info
W	- Print Window
Z	- Display End of File Pointer
1	- Beginning of File
9	- End of File
?	- Help Screen
↑	- Page Scroll Up (+)
↓	- Page Scroll Down (-)
TAB	- Setting Tabs

APPENDIX F: Shell/Monitor Command Summary
Command Description

ASSM <pathnm>	Assemble the stipulated file
BATCH <pathnm>	Perform monitor/shell commands from a text file
BLOAD <pathnm>	Load the stipulated MCL file, BRK to the GS monitor
BRUN <pathnm>	Load and execute the stipulatedMCL file
CATALOG <pathnm>	List the specified directory
CONTROL-Y<CR>	Return to Monitor/Shell from monitor
COPY	<file1> to <file2>
CREATE <pathnm>	Create a directory file withthe name <pathname>
DELETE <pathnm>	Delete stipulated file from the directory
EDIT [<pathnm>]	Invoke the text editor. If stipulated, load <pathname> to edit
FORMAT <Volume>	Initialize specified volume
HELP	Display the monitor/shell commands and descriptions
HOME	Clear the screen, home the cursor
LIST <pathnm>	List the stipulated text file to the screen
LOCK <pathnm>	Protect the stipulated file
ONLINE	Display all online volumes
PREFIX <Pathnm>	Set or determine default prefix
QUIT	Perform a ProDOS 16 quit
RENAME	<file1> to <file2>
UNLOCK <pathnm>	Unprotect the stipulated file

APPENDIX G: IIGS Monitor Usage

This is a summary of the most used APPLE IIGS system monitor commands. For more detailed information reference *APPLE IIGS technical reference manual* ISBN 0-07-881009-4 published by OSBORNE McGRAWHILL.

This is a example of how an address is input to the monitor.

The 12/ sets the program bank to 12 HEX.

The 34FF portion is the address within the 12th bank accessed.

Example:

```
*12/34FF Return
```

DISASSEMBLE LIST

Follow the address with a capital L.

0/1234L Return will give a disassembly listing starting at address 1234 in bank 0 for the length of one screen.

Example:

```
*0/1234L Return
```

Entering 'L' Return will continue the display.

DISPLAY MEMORY (DUMP)

Displays memory as from 100 to 200 in bank 0.

Example:

```
*0/100.200 Return
```

MODIFY CONSECUTIVE MEMORY

Key the address followed by a colon or semicolon then data. The exam-

ple below will modify bank 2 locations 0000, 0001 and 0002 respectively.

Example:

```
*2/0000:A5 BB 00 Return
```

EXECUTE (GO)

To execute a program in memory enter the bank and address

Example:

```
*3/0010X Return
```

STEP (Not yet supported)

To step the program one instruction at a time enter S after the address.

Example:

```
*4/0010S Return
```

TRACE (Not yet supported)

To run a program in TRACE enter T after the address.

Example:

```
*5/0040T Return
```

CHANGING REGISTERS

Note: Characters used are case sensitive !

CHANGE (A) REGISTER (val) = A

CHANGE (X) REGISTER (val) = X

CHANGE (Y) REGISTER (val) = Y

CHANGE (D) REGISTER (val) = D

CHANGE DATA BANK REGISTER (val) = B

CHANGE PROGRAM REGISTER (val) = K

CHANGE STACK POINTER (val) = S

CHANGE PROCESSOR STATUS (val) = P

CHANGE MACHINE STATE (val) = M

CHANGE QUAGMIRE STATE (val) = Q

CHANGE ACCUMULATOR MODE (val) = m

CHANGE INDEX MODE (val) = x

CHANGE EMULATION MODE (val) = e

CHANGE LANGUAGE CARD BANK (val) = L

GLOSSARY

6502 ADDRESSING FORMAT: Two byte addresses specified in least significant byte, most significant byte order.

6502 MICROPROCESSOR: CPU used in the Apple IIe, Apple][plus and Apple][.

65C02 MICROPROCESSOR: CPU used in the enhanced Apple IIe, new Apple IIe, and Apple IIc, software written for the 6502 will run on it. This chip has 27 additional machine language instructions.

65816 MICROPROCESSOR: CPU used in the Apple IIGS and Apple IIe Upgraded GS, software written for the 6502 and 65C02 will run on it.

ABSOLUTE ADDRESSING: Generally refers to addresses greater than $00FF (255) and less than $10000.

ALPHANUMERIC: Usually used to describe characters which consist of letters of the alphabet and digits.

ASCII code: ASCII is the acronym for American Standard Code for Information Interchange. A standardized code used to represent letters, digits and punctuation symbols. Apple uses this code but sets the most significant bit when used under DOS 3.3. The capital letter A is 65 (decimal) in the code.

ASSEMBLER: A program which can take as input an assembly language text file and translate it into the binary code the computer can execute. It usually gives additional information.

ASSEMBLY SOURCE CODE: A formatted text file an assembler can process into binary code.

ASSEMBLY LANGUAGE: The lowest programming language, specific to a given microprocessor, that uses short mnemonics corresponding directly to machine instructions and that allows a programmer to use symbolic code. At this level, the programmer is directly programming the CPU.

BINARY CODE: A numbering system consisting only of zeroes and ones (base 2).

BINARY FILES: Machine language programs saved on disk or tape.

BIT: Acronym of Binary digit. The smallest unit of information in a computer that can be represented by a zero or a one.

BRANCHING: Causes the program to begin execution at another memory location. The 65816 use relative addressing with branching. See JUMP.

BREAK POINT: Used in machine language debugging. When executed, it will cause the system to dump all registers, flags and counters and halt execution. The binary code on the 65816 is a zero (0).

BYTE: A collection of bits wired together. In most cases, a byte consist of eight bits. A byte can represent a character.

CPU: Stands for Central Processing Unit, the "brain" of the computer. When writing in machine language, you are actually programming the CPU.

CHAINING: The process of linking separate text files by the compiler. The compiler can successfully compile separate text files, as though they were a whole program.

COMPILER: A program that converts a program, usually a text file, written in a high-level language into either machine code or assembly language

CONTROL PANEL: A ROM based ancillary program that controls the functioning of slot and ports of the Apple IIGS.

CURSOR: A special character, often blinking, used to show the user where he will be entering characters on the screen.

DECIMAL: A numbering system based on the number 10. The numbering system we use in everyday life.

DIRECT ADDRESSING: Consists of either direct page addressing or absolute addressing.

DISASSEMBLER: A program which takes the binary numbers stored in the computer and translates then into assembly-like code.

EDITOR: A program which allows the user to create, modify and save text files.

FLAG: A boolean variable which can be set, so that later a determination can be made based on its value.

FILE: A collection of data stored in some memory device. This can be the computer's memory, a disk or a tape. On magnetic media, a file name is usually associated with the file.

HEXADECIMAL: A number system based on the number 16 (base 16). Numbers Ø through 9 and letters A through F are used. The letters represent decimal numbers 1Ø through 15.

IMMEDIATE ADDRESSING: Addressing mode in which the byte(s) following the op code contains the value to be used. LDA #$FØ will cause the accumulator to load an $FØ value.

INCLUDING: Inserting a file contained on disk as if it were physically sitting at the include statement. INS is this system's include reserved word.

INDIRECT ADDRESSING: Addressing mode in which the specified address contains the address which will be used.

JUMP: Causes the program to begin execution at the specified location. Varies from branch in that it uses absolute rather than relative addressing.

LABEL: Used in assembly and most higher level languages to allow the programmer to reference a part of the program. In assembly language, the label will stand for an address in memory.

LOAD: The act of bringing information from some long-term storage device such as disk to the computer's memory.

MACHINE CODE: Almost synonymous with assembly code. Usually refers to the binary code which the computer directly executes.

MACRO: In assembly language, a segment of text which can be later referenced and thereby inserted as part of the code. Usually accepts parameters.

MCL FILES: Static or relocatable load files generated by the Micol Macro™ assembler.

MEMORY LOCATION: The same as a byte. Can be thought of as a little box in the computer containing a piece of information.

MEMORY MANAGER: A ROM based program that supervises the use of the computer's memory.

MICOL SYSTEMS: A dynamic software house founded almost simultaneously in Southern California and Ontario, Canada in 1983. Dedicated to quality systems software, MICOL is the acronym of Micro Computer Languages.

MNEMONIC: A collection of characters which can help you remember something. 'JMP' stands for jump and represents $4C in machine code and is a mnemonic for it.

MODULARIZATION: The act of breaking your program into small, easily maintainable parts. While little overhead is involved, it greatly minimizes the maintenance costs.

MONITOR: A program which interfaces the human with the machine code in his computer.

MONITOR/SHELL: The human interface portion of Micol Macro™.

OCTAL: A number system based on the number 8 (base 8). Digits 0 through 7 are used. A 10 in octal is decimal 8.

OP CODE: Short for operation code, the second field in a 6502/65C02, 65816 assembly line which instructs the CPU what action to take.

OPERAND: The address field following the op code.

PASS 1: In an assembler, the phase in which all addresses are resolved.

PASS 2: In an assembler, the phase in which the code is generated.

PROGRAM: A collection of instructions designed to perform (a) specific action(s).

PSEUDO OP CODE: Instructions which resemble operation codes, but are usually designed to instruct the assembler what action to take.

RADIX: The base value of a numbering system. The radix of the decimal system is 10.

REGISTERS: Memory locations within the CPU having special features not

found in memory. Without registers, your computer would be worthless. In the 6502/65C02 microprocessor, the 5 registers are:

A accumulator — virtually all mathematics are performed here.

X register — mainly used for indexing

Y register — mainly used for indexing

Status register — condition flags based upon certain operations are kept here

Stack pointer — points to location in page one

The 65816 has additional registers. They are:

Data Bank Register: The bank number from which data is usually accessed.

Program Bank Register: The bank portion of the program counter.

Direct Page Register: The 16 bit register which points to the beginning of direct page.

Accumulator B: The MSB of the 16 bit accumulator.

In addition, there are the memory select and index register select bits in the status register as well as a shadow emulation bit.

SAVE: The act of storing all or part of a computer's memory to some long-term storage device.

STATUS FLAGS: Bits within the status register which are set or unset by certain conditions. The status flags in the status register are: zero, sign, memory select, decimal, index register select, interrupt, overflow, break and carry. All "decisions" are based upon the status (0 or 1) of these flags.

STRING: A collection of characters. The 'STR' pseudo op is used by the Micol Systems' assembler to declare strings, e.g. 'THIS IS A STRING'.

STRUCTURED PROGRAMMING: A systematic approach to the creation of software by using a step-by-step procedure for solving the problem. It consists of a smooth program flow, modularization of code, etc.

TOGGLE: To change state from on to off and back again.

ZERO PAGE: The area in memory between locations 0 and 255 in bank zero. The zero page is extended to direct page on the 65816 and can occupy 256 bytes anywhere in bank 0.

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

Z

Special Characters